Marko Keronen

Käytöspuut ja niiden käyttö taistelupelin tekoälyssä

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Luonnontieteiden ala
Tietojenkäsittelyn koulutusohjelma
Syksy 2013

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittely
Tekijä(t) Marko Keronen	
Työn nimi Käytöspuut ja niiden käyttö taistelupelin tekoälyssä	
Vaihtoehtoiset ammattiopinnot Peliohjelmointi	Toimeksiantaja -
Aika Syksy 2013	Sivumäärä ja liitteet 59

Käytöspuut ovat vielä tuore, mutta suuren luokan kaupallisissakin peleissä käytetty tekoälytekniikka, jonka avulla hallitaan monimutkaisuutta. Tämän opinnäytetyön tavoitteena on esitellä käytöspuut tekniikkana, määritellä toimivan käytöspuujärjestelmän kehityksen periaatteita, ja tarkastella käytöspuiden soveltuvuutta taistelupelin tekoälyssä.

Tekoälyllä voidaan tarkoittaa laajasti erilaisia asioita, ja tekoälyillä voi alasta riippuen olla täysin toisistaan poikkeavia tavoitteita. Peleissä tekoälyn tavoitteena on kuitenkin aina viihdyttää pelaajaa. Hauskan tekoälyn luomiseen on useita keinoja, ja tekoälyn viihdyttävyys onkin tiukasti sidoksissa kaikkiin pelin ominaisuuksiin ja järjestelmiin. Taistelupelit ovat tekoälyn kannalta erityisen haastava peligenre, sillä suuret hahmomäärät ja hahmojen yksilölliset liikkeet vaativat, että tekoälyn toimintaa voidaan hallita aina pienintä yksityiskohtaa myöten.

Käytöspuujärjestelmän toteutukseen voidaan soveltaa perusperiaatteita, joiden avulla järjestelmästä saadaan selkeä ja toimiva. Täysin uudelleenkäytettävät perusosat, osien hallintaa auttava tukijärjestelmä ja sopivilla osilla toteutettu hierarkia tekevät tekoälyn käytösmallien luomisesta sujuvaa. Lisäämällä käytöspuujärjestelmän yhteyteen sen kehitystä tukevan seurantajärjestelmän, saadaan järjestelmässä piilevät virheet löydettyä helpommin ja nopeutettua itse kehitysprosessia.

Opinnäytetyössä toteutettiin yksinkertaisen tekoälyn pohjaksi käytöspuita käyttävä tekoälyjärjestelmä taistelupeliin. Tekoälyjärjestelmän rinnalle rakennettiin seurantajärjestelmä avustamaan kehitystä. Jo yksinkertaisenkin tekoälyn toteuttaminen on äärimmäisen työlästä, ja viihdyttävän ja virheettömän tekoälyn rakentaminen on vähillä resursseilla miltei mahdotonta. Käytöspuilla rakennettu tekoälyjärjestelmä on kuitenkin tehokas tapa hallita yksilöllistä logiikkaa, ja toimii pelityypistä riippumatta.

Kieli	Suomi
Asiasanat	Ohjelmointi, peliohjelmointi, taistelupelit, tekoäly, videopelit
Säilytyspaikka	☐ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Marko Keronen	
Title Behavior Trees and Their Use in Fighting Game AI	
Optional Professional Studies Game programming	Commissioned by -
Date Fall 2013	Total Number of Pages and Appendices 59

Behavior trees are a fresh technique when creating game artificial intelligence, yet, employed even in large scale commercial games to handle complexity. The aim of this thesis is to explore behavior trees as an AI technique, define principles for developing a solid behavior tree system and determine if behavior trees can be applied to create a fighting game AI.

Depending on the field, artificial intelligence can mean vastly different concepts and have wildly varying goals. In games, however, the AI is meant to entertain the player. There are numerous ways to create an enjoyable AI, and the level of fun is closely tied to all of the features in the game. Fighting games are particularly challenging in regards to AI, as their large numbers of characters and highly individual moves require precise customization from the AI system.

Basic principles can be applied when implementing a behavior tree system to ensure a clear and solid structure. Modular components, a support framework to handle control flow and a hierarchical structure as the basis of the system help when creating AI behaviors. With an attached diagnostics system, errors in the behavior tree system can be located more easily and the development process made more effective.

As the practical part of the thesis, a behavior tree system was built as a basis for a simple fighting game AI. A diagnostics system was implemented to assist in the AI development. Creating even the simplest AI requires a tremendous amount of work, and creating an entertaining AI that functions without error is nearly impossible with scarce resources. Nevertheless, behavior trees are an effective way to handle highly individual AI logic and can be applied to any AI system regardless of genre.

Language of Thesis	Finnish
Keywords	Artificial intelligence, fighting game, game programming, programming, video game
Deposited at	☐ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

SISÄLLYS

SYMBOLILUETTELO

Abstrakti luokka	Luokka, jota ei voida käyttää vaan sellaisenaan, mutta josta perityt luokat voivat käyttää sen ominaisuuksia.
Animaatio	Sarja peräkkäin näytettäviä kuvia, jotka saavat aikaan vaikutelman liikkeestä.
Animaatiokehys	Yksittäinen animaatiossa oleva kuva, joiden sarjasta animaation liike muodostuu.
Lippu	Bool-tyyppinen muuttuja, jota käytetään merkitsemään onko jokin tietty ominaisuus päällä.
Nimiavaruus	Ohjelmoinnin yksikkö, jonka sisällä olevat luokat pystyvät käyttämään toisiaan.
Reitinhaku	Tekoälyyn usein liitettävä tapa, jolla jokin tekoälyhahmo etsii reitin kohteeseensa.
Skripti	Yleensä hieman ohjelmointikoodia muistuttava yksinkertaistettu sarja komentoja, joilla voidaan ohjeistaa ohjelman toimintaa.
Syöte	Esimerkiksi näppäimistöstä tai hiirestä saatava tila, joka tuotetaan laitteita käyttämällä. Näppäimistön tietyn näppäimen painaminen tuottaa syötteenä kyseisen näppäimen.
Törmäystarkastelu	Koodi, joka tarkastelee kappaleiden, kuten eri hahmojen, välisiä törmäyksiä.
Virtuaalinen	Luokan jäsenen tyyppi, joka kertoo, että luokan jäsen voidaan ylikirjoittaa perityissä luokissa, ja jonka perittyä muotoa käytetään alkuperäisen muodon yli.

1 JOHDANTO

Käytöspuut ovat suhteellisen uusi tekoälytekniikka. Siitä huolimatta ne ovat löytäneet tiensä jättimäisten budjettien AAA-peleihin asti, aina Halo 2:n ja Sporen kaltaisia huippunimiä myöten. Erityisesti käytöspuita on käytetty ensimmäisen persoonan ammuntapeleissä, joissa niiden tuomat hyödyt tulevat parhaiten esille. Uudelleenkäytettävyyttä parantavat ja tilojen hallintaa tuovat rakenteet ovat tehneet käytöspuista erinomaisen korvaajan yhä monimutkaiseksi käyvien tilakoneiden tilalle, koska ne eivät kärsi samalla tavalla suureksi kasvavasta tekoälyn logiikan määrästä.

Taistelupelit ovat tekoälyn kannalta yksi tarkinta hienosäätöä vaativimmista peligenreistä. Suuret määrät erilaisia hahmoja ja lukuisat määrät toinen toistaan erikoisempia hyökkäyksiä vaativat tekoälyltä laajamittaista reagointikykyä erilaisiin tilanteisiin. Samalla eri hahmojen tekoälyn voimakkuuden tulee pysyä mahdollisimman hyvässä tasapainossa keskenään, ja pystyä tarjoamaan sopivaa haastetta täysin eri tasoisille pelaajille, asettaen äärimmäisen tarkkoja vaatimuksia tekoälyn säädettävyydelle.

Tilakoneet ovat juuri hienosäätömahdollisuuksiensa ansiosta yleisesti taistelupeleissä käytetty tekniikka. Opinnäytetyön tavoitteena on selvittää, toimivatko käytöspuut myös taistelupeligenressä varteenotettavana vaihtoehtona tilakoneille. Työn päätavoitteena ei ole tehdä äärimmäisen monimutkaista, tai millään tavalla täysin valmista taistelupelin tekoälyä, vaan tekoälyjärjestelmä, joka toimii ennemminkin pohjana todellisen tekoälyn luomiselle.

Opinnäytetyössä tehtiin käytöspuilla tekoälyjärjestelmä Access Lumia –taistelupeliin. Käytöspuujärjestelmää käytettiin luomaan peliin yksinkertainen tekoäly, josta pyrittiin saamaan viihdyttävä vastustaja ihmispelaajalle. Käytöspuujärjestelmän tueksi rakennettiin myös seurantajärjestelmä, jota käytettiin apuna käytöspuujärjestelmän kehittämisessä ja testaamisessa.

2 TEKOÄLYN PERUSTEET

Tekoäly on käsitteenä häilyvä, sillä termin ympärillä on lukuisia koulukuntia, joiden näkemykset voivat erota toisistaan valtavasti (Tozour 2002 a, 5). Akateemisessa mielessä tekoäly voidaan määritellä esimerkiksi ihmismäisesti ja rationaalisesti käyttäytyvien ja ajattelevien tietokoneohjelmien tekemiseksi. Tämä määritelmä sisältää rationaalisen älykkyyden lisäksi myös muita ihmismäisiä ominaisuuksia, kuten toisten puolesta uhrautumista, jotka voidaan yhä käsittää älykkäänä niiden rationaalisuudesta välittämättä. (Schwab 2004, 4.)

Pelien tekoäly poikkeaa kuitenkin selvästi akateemisesta tekoälystä. Akateeminen tekoäly pyrkii erityisesti ymmärtämään sitä, kuinka tekoäly päätyy johonkin ratkaisuun. (Schwab 2004, 9.) Peleissä sen sijaan on tärkeämpää itse ratkaisun saavuttaminen. Tekoälyn täytyy onnistua halutussa tehtävässä niin, että se vaikuttaa tekevän järkeviä päätöksiä, välittämättä siitä miten päätöksiin on saavuttu. Peliä pelaavan ihmisen kannalta on täysin yhdentekevää minkälaisia tekniikoita tekoäly pinnan alla käyttää, kunhan sen käyttäytyminen näyttää uskottavalta. (Schwab 2004, 4.) Tekoälyn ei välttämättä tarvitse edes olla millään tavalla älykäs, tai sitä ei tarvita ollenkaan, niin kauan kuin pelaaja kuvittelee, että tietokoneen ohjaama vastustaja tekee jotain harkittua ja järkevää (Lidén 2004, 41).

2.1 Tekoälyn hauskuus

Sen lisäksi, että pelien tekoäly tähtää yleensä tuloksellisuuteen, tuo halutun tuloksen luonne vielä enemmän eroa akateemiseen maailmaan. Peleissä tekoälyllä on käytännössä vain yksi tarkoitus: tekoälyn täytyy tehdä pelaamisesta hauskaa. (Tozour 2002 a, 9.) Koska ihmiset eivät ole koskaan täydellisiä, oletetaan tekoälynkin usein tekevän virheitä ja valitsevan ratkaisuja, jotka eivät välttämättä tähtää parhaaseen mahdolliseen lopputulokseen - joka akateemisessa maailmassa on usein tekoälyn päämääränä. Jos esimerkiksi ensimmäisestä persoonasta kuvatussa ammuntapelissä, kuten Quakessa, tekoälyvastustaja pelaajan kohdatessaan tähtäisi ja väistäisi aina täydellisesti, olisi pelin pelaaminen turhauttavaa. (Schwab 2004, 10.) Monissa pelityypeissä täysin ylivoimaisen tekoälyn ohjelmointi onkin varsin yksinkertaista, ja vaikka tekoälyä voidaan tällöin ajatella älykkäänä, ei se toteuta millään tavalla tarkoitustaan (Tozour 2002 a, 9).

Koska hauskuus on pelien tekoälylle kaikki kaikessa, on tärkeää tietää mikä tekee tekoälystä - tai oikeastaan koko pelistä - viihdyttävän. Hauskuuden luominen peleihin on lukuisien osien summa, mutta keskeisesti pelin täytyy olla yhtä aikaa riittävän palkitseva ja sopivan haastava. Käytännössä nämä kaksi asiaa ovat myös yhteyksissä toisiinsa, sillä liian anteliaasta erilaisia palkintoja pelaajalle antava peli voi tehdä pelistä liian helpon, kun taas pelaajan taitoja äärirajoilla käyttävän tilanteen läpi pääseminen voi tuntua hyvinkin palkitsevalta. Pelin vaikeustason kanssa pitää kuitenkin olla hyvin tarkkana, sillä pelaaja luovuttaa helposti, jos kohdattu tilanne tuntuu mahdottomalta läpäistä, ja vastavuoroisesti ei pidä peliä viihdyttävänä, jos se ei tuota hänelle tarpeeksi vastusta. (Schwab 2004, 552.)

Koska liian hankala vaikeustaso ei palvele pelin tarkoitusta, voi usein olla tarpeellista tehdä tekoälystä älykkäämmän sijasta hieman tyhmempi. Tekoäly ei saa kuitenkaan olla huonosti suunniteltu, vaan se voidaan laittaa tekemään virheitä tarkoituksella. Näin tekoälyn tekemät virheet voidaan säätää näyttämään uskottavilta, ja sen toiminta saadaan näyttämään pelaajan näkökulmasta järkevältä ja inhimilliseltä. Vielä pidemmälle vietynä tekoälyjärjestelmässä olevat oikeat, loogiset virheet voidaan valjastaa hauskojen tilanteiden luomiseen, mikäli virhetilanteet pystytään tunnistamaan. Esimerkiksi Valven Half-Life-pelissä tekoälyn hallitsemat hahmot eivät aina löytäneet pakoreittiä, kun pelaaja heitti kranaatin niiden lähelle. Ongelma saatiin tunnistettua, ja tekoälyhahmot komennettiin panikoimaan laittamalla ne menemään kyykkyyn kädet päänsä päällä, jos pakoreittiä ei löytynyt. Näin virhettä ei tarvinnut edes korjata, ja käytös loi hahmoille selkeän luonteenpiirteen. (Lidén 2004, 41 - 48.)

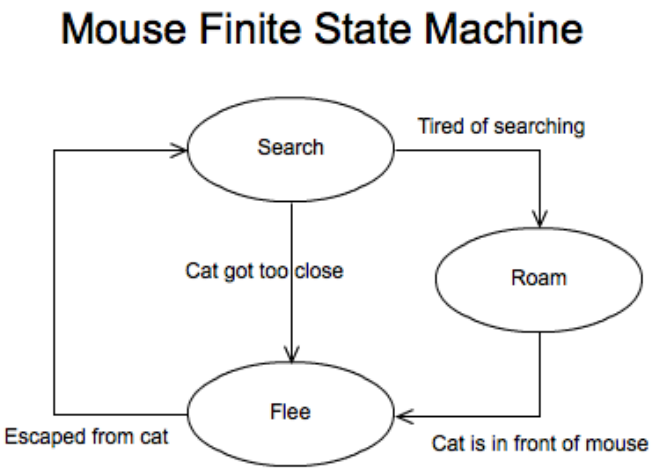
Joskus äärimmäisen hankala vaikeustaso ei kuitenkaan tee pelistä vähemmän viihdyttävää, sillä yksi tärkeistä viihdyttävyyttä luovista asioista on uutuudenviehätys. Ominaisuudet, jotka tuntuvat tuoreilta ja yllättäviltä, voivat muuttaa rasittavatkin tilanteet hauskoiksi. Esimerkiksi pelissä Medal of Honor tekoälyhahmot osasivat poimia pelaajan niitä kohti heittämät kranaatit ja heittää ne takaisin pelaajaa kohti. Monet peliä pelanneet kehuivatkin siksi pelin tekoälyä todella erityiseksi tapaukseksi, vaikka tekoäly saattoikin saada ominaisuudella pelaajan päihitettyä. (Schwab 2004, 552 - 553.)

2.2 Yleisiä tekoälytekniikoita

Tässä kappaleessa esitellään kolme tekoälytekniikkaa, jotka ovat yleisesti käytössä ja ovat erityisen merkittäviä käytöspuiden ja taistelupelien tekoälyn ymmärtämisessä. Nämä tekniikat ovat rajallinen tilakone, sen hierarkinen muoto, joka on samalla käytöspuiden esiaste, sekä erilaiset skriptausjärjestelmät.

2.2.1 Rajallinen tilakone

Rajallinen tilakone (finite state machine) on tekoälyrakenne, joka sisältää joukon tiloja (state) ja niiden välisiä siirtymiä (transition) (Bailiee-de Byl 2004, 234 – 235). Tilakoneen tilat rakentuvat toiminnoista, joita tekoälyhahmo suorittaa kyseisessä tilassa ja tilalle ominaisista siirtymistä, joiden avulla hahmo siirtyy uuteen tilaan (Kuvio 1). Siirtymät tarvitsevat myös ehtoja, joiden täyttyessä tilasiirtymä suoritetaan. (Champandard 2007 a.)



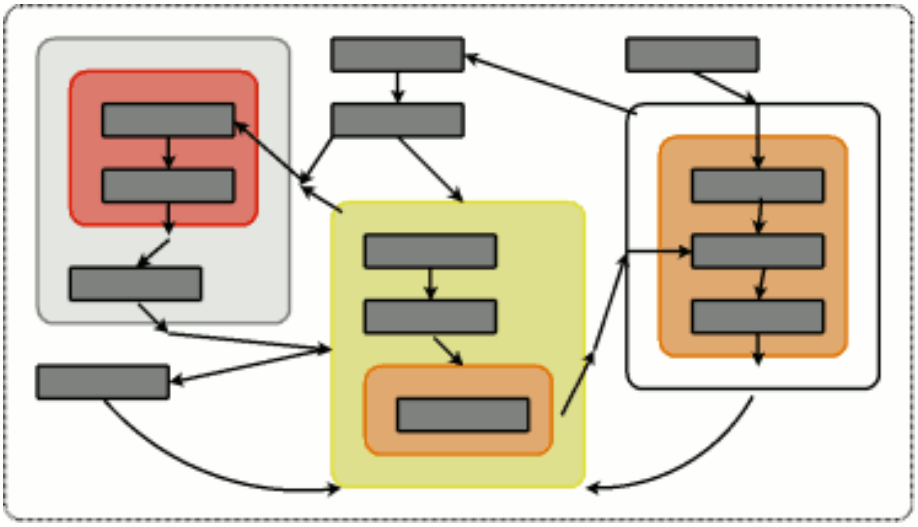
Kuvio 1. Kissalta pakenevan hiiren tilakone, jossa näkyvät hiiren tilat ja tilojen väliset siirtymät (Tuttle 2011)

Rajallisten tilakoneiden keskeisenä ongelmana on niiden huono laajennettavuus. Tiloja ei voida yleensä käyttää kuin siihen yhteen nimenomaiseen tarkoitukseen, johon se on rakennettu. Tila pystyy siis yleensä ratkaisemaan vain yhden ongelman, ja senkin vain tietynlaisessa tilanteessa. Kun jokainen ongelma vaatii oman tilansa, kasvaa myös tilojen välisten siirtymien

määrä. Lopulta tilakoneesta voi tulla niin monimutkainen, että sen hallitsemisesta tulee miltei mahdotonta. (Champandard 2007 a.)

2.2.2 Hierarkinen rajallinen tilakone

Hiearkinen rajallinen tilakone (hierarchical finite state machine) pyrkii vähentämään rajallisten tilakoneiden siirtymien määrän lisääntymisen ongelmia. Hierarkisessa rajallisessa tilakoneessa tilat ryhmitellään ylätiloihin (super-state), jotka jakavat keskenään siirtymälogiikkaa (Kuvio 2). Täten ryhmällä voi olla yksi yleistetty siirtymätapa (generalized transition), sen sijaan että ryhmän jokaisella jäsenellä olisi sama logiikka erikseen. (Champandard 2007 b.)



Kuvio 2. Hierarkinen rajallinen tilakone. Tiloille määritellyt ylätilat on kuvattu laatikoin. (Champandard 2007 b.)

Hierarkisen rajallisen tilakoneen ongelmat tulevat vastaan toteutuksessa. Eri tilanteissa uudelleenkäytettävien siirtymätapojen suunnittelu voi olla hyvin haastavaa, ja siten viedä paljon aikaa. Siirtymiä joudutaan edelleen myös muokkaamaan käsin, joka voi käydä laajemmissa käyttötapauksissa työlääksi. (Champandard 2007 b.)

2.2.3 Skriptaus

Sriptauksella (sripting) tai skriptauskielellä (scripting language) tarkoitetaan yksinkertaistettua ohjelmointikieltä, jonka avulla voidaan kasata pelin tekoälylle logiikkaa tai käytösmalleja (Schwab 2004, 337). Skriptaamalla pyritään aina yksinkertaistamaan monimutkaisia toimintoja ohjelmistossa, ja siten skriptauskielet ovatkin erittäin projektikohtaisia. Se, mitä milläkin skriptauskielellä on mahdollista toteuttaa, on usein täysin riippuvaista projektin tarpeista. (Berger 2002, 505.) Tämän takia skriptauskielet voivatkin olla hyvin yksinkertaisia kieliä, joita ohjataan yksinkertaisilla avainsanoilla, tai kokonaisia, täysin valmiita ohjelmointikieliä. (Schwab 2004, 337.)

Useimmat tekoälyjärjestelmät toimivat miltei täysin koodissa, ja niiden säätämiseen tarvitaankin siten yleensä aina ohjelmointikokemusta. Kun pelin tekoälyhahmojen käyttäytymisen luominen vaatii huomattavia määriä luovaa työtä, on soveliaampaa - tai jopa välttämätöntä - että pelisuunnittelijat pääsevät käsiksi tekoälyn toimintoihin. Skriptauskielet kehitetäänkin yleensä pelisuunnittelijoita varten. Näin suurempi määrä ihmisiä voi työskennellä luovaa työtä vaativan sisällön parissa, ja ohjelmoijat voivat keskittyä muihin tehtäviin. (Schwab 2004, 337.)

Suurimpia etuja skriptauksen käytössä pelin tekoälyjärjestelmässä onkin se, että vähemmän teknistä osaamista hallitsevat ihmiset voivat muokata tekoälyn käyttäytymistä hyvin laaja-alaisesti. Skriptikielen yksinkertaisuus tekee tekoälyn muokkaamisesta myös helpommin lähestyttävää, jolloin kielen käyttäjät saadaan motivoitumaan paremmin rakentamaan erilaisia järjestelmiä sen avulla. Skriptauksella voidaan myös tehdä tekoälyn arvoja säätämällä nopeita testejä - tai prototyyppejä - erilaisista käyttäytymismahdollisuuksista, mikäli skriptausjärjestelmä mahdollistaa skriptitiedostojen uudelleenlataamisen pelin ajon aikana. (Schwab 2004, 360 - 361.)

Skiptauskielen tekeminen ei kuitenkaan koskaan ole aivan yksiselitteistä, eikä sellaisen käyttäminen tuo mukanaan vain hyötyjä (Berger 2002, 505). Skriptien suoritus on esimerkiksi merkittävästi hitaampaa kuin puhtaan konekieleksi käännetyn koodin suorittaminen. Skriptausjärjestelmä on myös ennen kaikkea itse pelistä käytännössä täysin erilainen tuote, jota pelaajien sijasta käyttävät pelin kehittäjät, ja siten sen käyttöliittymältä vaaditaan aivan erilaisia asioita. Järjestelmän toteuttaminen lisää välttämättä työmäärää. Sitä täytyy ylläpitää, siihen voidaan joutua lisäämään ominaisuuksia ja sen käyttöä joudutaan opettamaan muille kehittä-

jille. Kun ominaisuuksia lisätään enemmän, voi skriptityökaluista myös tulla liian monimutkaisia - joka poistaa helposti koko järjestelmän hyödyllisyyden, erityisesti vähemmän teknistä osaamista omaavien ihmisten kannalta. (Schwab 2004, 362 - 364.)

Skriptauskielen toteuttaminen juuri vähemmän teknisille ihmisille onkin osasyynä kaikkein yleisimmille ongelmille. Skripteissä olevia ongelmia voi olla hyvin vaikea selvittää, sillä esimerkiksi pelisuunnittelijat ja muut luovaa työtä tekevät eivät usein hallitse yhtä vahvoja ongelmanratkaisu- ja testaustaitoja kuin ohjelmoijat, kun kyseessä on koodissa oleva ongelma. Ohjelmoijille nämä taidot kehittyvät yleensä pitkän ajan kuluessa, joten useat ohjelmointia osaamattomat eivät välttämättä osaa edes koodin testauksen perustekniikoita. Lisäksi skriptauskieliin rakennettaan harvoin yhtä voimakkaita ongelmanratkaisutyökaluja kuin valmiista ohjelmointikielistä löytyy, koska toteuttaminen toisi lisätyötä kielen kehittäjille. (Schwab 2004, 362.)

3 TAISTELUPELIT

Pelit, joissa pääasiassa on nyrkeillä, miekoilla tai muilla lähikontaktiin pyrkivillä aseilla taistelu, ovat ajan kuluessa jakaantuneet useisiin eri leireihin. Beat-'em'up-peleissä pelaaja asetetaan usein suurta vihollisjoukkoa vastaan, ja yksittäinen vihollinen on pelaajaan nähden miltei voimaton. Taistelupelit sen sijaan keskittyvät tuomaan toimintaan intensiivisyyttä kahdenkeskisessä turnausmuotoisessa tappelussa, jossa kumpikin osapuoli on lähtökohtaisesti kyvyiltään samalla tasolla. (Spencer 2008.) Kolmanneksi osapuoleksi ovat taistelupeleistä erottuneet vielä urheilua mallintavat pelit, kuten paini- ja nyrkkeilypelit (Hardcore Gaming 101).

1990-luvun alkupuolella Capcomin julkaisema Street Fighter 2: The World Warrior toi taistelupelit näkyväksi genreksi suurella suosiollaan, ja toi mukanaan useita tärkeitä ominaisuuksia (Schwab 2004, 203). Pelissä pelaaja pystyi esimerkiksi valitsemaan kahdeksasta eri pelattavasta hahmosta, joka oli tuolloin mittava määrä, etenkin verrattuna pelisarjan ensimmäisen osan kahteen valittavaan hahmoon. Hahmot myös erosivat toisistaan selkeästi yksilöllisillä liikkeillään, ja olivat kaiken lisäksi voimakkuudeltaan keskenään erinomaisessa tasapainossa. (Houghton 2012.)

Street Fighter 2 toi mukanaan myös jatkossa koko peligenreä määrittelevän ominaisuuden: liikesarjat. Liikesarjassa pelaaja voi yhden hyökkäyksen jälkeen siirtyä toiseen hyökkäykseen miltei tauotta. Tuloksena on usein hyökkäysten sarja, jota ei voida torjua, mikäli sarjan ensimmäinen hyökkäys on osunut kohteeseen. (Houghton 2012.) Liikesarjojen keskeisenä osana olivat myös voimakkaat erikoishyökkäykset, joita tehtiin yleensä monimutkaisilla näppäinpainallusten sarjoilla. Erikoishyökkäyksien opettelusta tuli nopeasti yleinen toimenpide pelin pelaajien keskuudessa. (1UP.)

3.1 Taistelupelin tekoälyn edellytykset

Taistelupeleissä eri hahmojen täytyy olla hyvässä tasapainossa keskenään, koska se on usein ollut genren sisällä suurimpia myyntivaltteja. Tekoälyn tasapainotusta varten sitä täytyy siis voida hallita erittäin tarkasti ja yksityiskohtaisesti. Käytännössä kehittäjän täytyy päästä käsiksi yksittäisen liikkeen yksittäiseen animaatiokehykseen (animation frame), ja pystyä määrittämään animaatiokehyksen kohdalla soitettavia ääniä, törmäystarkasteluun käytettäviä muotoja tai mitä tahansa, jota kehyksen aikana voi tapahtua. (Schwab 2004, 205.)

Törmäystarkastelu, etenkin hahmojen välinen törmäystarkastelu, onkin taistelupeleissä hyvin keskeisessä osassa. Törmäystarkasteluun käytetty järjestelmä on yleensä hyvin monimutkainen. Tyypillisesti, hahmoille määritellään alueita törmäyksiä varten, ja alueet saattavat muuttaa muotoaan, hävitä tai ilmestyä minkä tahansa animaatiokehyksen aikana. Myös törmäyksien aiheuttamat liiketilat ovat ennalta määriteltyjä, eivätkä perustu todelliseen fysiikkaan. (Schwab 2004, 205.)

3.2 Taistelupeleissä yleensä käytetyt tekoälytekniikat

Koska pelisuunnittelijoilta ei yleensä oleteta erityistä ohjelmointiosaamista, ovat tappelupeleihin valitut tekniikat yleensä pelin sisällön vaatiman datan ja pelimekaanikan vaatiman logiikan erottavia datapainoitteisia järjestelmiä (data-driven systems). Näin säätömahdollisuudet voidaan erottaa varsinaisesta ohjelmistokoodista. Samasta syystä monet taistelupelit käyttävät tekoälyssään skriptikieltä, joka on räätälöity juuri kyseisen pelin tarpeisiin. (Schwab 2004, 207.)

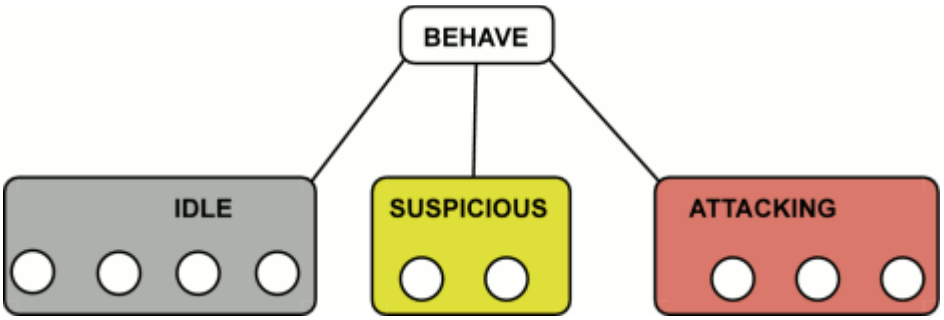
Taistelupelien vaatimukset tarkoista hienosäätömahdollisuuksita ja datapainoitteisista rakenteista asettavat rajoituksia niissä yleisimmin käytettäviltä tekoälytekniikoilta. Useimmin käytetäänkin yksinkertaisia, mutta täysin räätälöitävissä olevia malleja. Yleisiä tekniikoita ovat esimerkiksi rajalliset tilakoneet, jotka rakennetaan datapainoitteisina, tai erilaiset skriptatut järjestelmät, jotka voivat lisäksi auttaa välianimaatioiden ja liikesarjojen tekemisessä. (Schwab 2004, 207 - 208.)

4 KÄYTÖSPUUT

Tässä kappaleessa määritellään mitä käytöspuut ovat ja mitä etuja niissä on. Lisäksi käsitellään se, millä tavoilla käytöspuita voidaan käydä läpi. Kappaleessa esitellään myös miten käytöspuita on käytetty kahdessa tunnetussa suuren budjetin peleissä: Halo 2:ssa ja Sporessa.

4.1 Käytöspuun määrittely

Käytöspuut (behavior tree) ovat vieneet hierarkisuuden vielä pidemmälle kuin hierarkisessa rajallisessa tilakoneessa. Niissä hierarkisuuden tasoja voi olla rajaton määrä. Puun huipulta löytyy juurisolu (root node), jonka alle puu rakentuu. Jokaisella solulla, mukaanlukien juurisolu, voi olla rajaton määrä lapsisoluja (child node), joita lapset omistava solu hallitsee. Näitä hallitsevia soluja kutsutaan vanhemmiksi (parent node). Solua, jolla ei ole lapsisoluja, kutsutaan puun lehdeksi (leaf node), ja juuri lehtisolujen tehtävänä on yleensä varsinaisten toimintojen (action) suorittaminen. (Kirby 2010, 350 – 352.)



Kuvio 3. Yksinkertainen, uudelleenkäytettävistä osista koostuva käytöspuu (Champandard 2007 c)

Käytöspuut välttävät tilakoneiden ongelmat tekemällä tiloista itsenäisiä kokonaisuuksia – käytöksiä (behavior) (Kuvio 3). Käytökset ovat yksinkertaisuudessaan kokoelma suoritettavia toimintoja, joiden suoritusta ja suorituksen lopettamista voidaan hallita. Käytöksillä ei ole omaa siirtymälogiikkaa, vaan siirtymistä huolehtii käytöspuussa olevan solun (node) vanhempi (parent). Siirtymät määrittyvät siis automaattisesti riippuen siitä minkälaiseen yhteyteen käytös lisätään suoritusvaiheessa. (Champandard 2007 c.)

4.2 Käytöspuiden läpikäyminen

Käytöspuiden läpikäymiseen käytetään erityisesti kahta lähestymistapaa: ylhäältä alas kulkemista (top-down evaluation) tai alhaalta ylös kulkemista (bottom-up evaluation) (Kirby 2010, 351). Ylhäältä alas kuljettaessa puun käsittely aloitetaan puun juurisolusta. Kun solua käsitellään, käsitellään käytännössä koko solun alle muodostuva alipuu (sub-tree). Lopulta käsitelty solu palauttaa aina jonkin palautusarvon (return state), jonka mukaan puun tila määrittyy. Tyypillisiä solun palautusarvoja ovat esimerkiksi valmis (ready), jolloin solu voi vastaanottaa suorituspyynnön, onnistuneesti suoritettu (successful) tai epäonnistuneesti suoritettu (failed), jotka kertovat solun suorittamisen valmistumisesta, ja suorittumassa (running), jolloin solua suoritetaan edelleen ja sitä tulisi kutsua uudestaan seuraavalla päivityskierroksella (Knafla 2011). Solujen suoritus jatkuu niin kauan, kunnes jokin solu palauttaa jonkin suorituksen keskeyttävän signaalin (Kirby 2010, 351 – 352).

Ylhäältä alas kuljettaessa puun ylhäällä olevien solujen täytyy pystyä tarkasti ennustamaan haluaako jokin niiden lapsisoluista aktivoitua. Käytännössä ennustus täytyy tehdä aina käytöspuun lehtisoluihin asti. Tässä tapauksessa nimenomaan nämä toimintoja suorittavat lehtisolut ovat varsinaisia käytöksiä, ja kaikki niiden yläpuolella olevat solut ovat päättäjiä (decider). (Kirby 2010, 351 – 352.)

4.3 Käytöspuiden käyttökohteet

Käytöspuut ovat lyöneet itsensä läpi myös useissa äärimmäisen korkean luokan peleissä, jotka tunnetaan maailmanlaajuisesti. Tässä kappaleessa esitellään kuinka käytöspuita on sovellettu Bungien Halo 2 -pelin ja Maxisin Spore-pelin tekoälyjärjestelmissä. Esimerkit auttavat ymmärtämään minkälaisiin haasteisiin käytöspuilla voidaan vastata, ja minkälaisia periaatteita niitä kehittäessä kannattaa suosia.

4.3.1 Halo 2

Halo 2 on Bungien kehittämä ensimmäisestä persoonasta kuvattu ammuntapeli, jossa kolme täysin erilaista osapuolta - ihmiset, avaruusolioiden rotu Covenant ja parasiittimaisista olennoista koostuva Flood - taistelevat keskenään, ja pelaajan ohjaama sankari, Master Chief, on kaiken toiminnan keskellä (Kasavin 2004). Halo 2:n tekoälyssä yksi suurimmista vastaan tulleista haasteista oli tekoälyn monimutkaisuuden hallinta. Tekoälyhahmojen saaminen käyttäytymään oikeilla tavoilla oikeisiin aikoihin niin, että ne tuntuvat eläviltä olennoilta, ja lukuisien eri käyttäytymismallien toteuttaminen hahmoille asetti pelin tekoälyjärjestelmälle erityisiä vaatimuksia. (Isla 2005.)

Monimutkaisuuden hallitsemiseen Halo 2:een valittiin tekoälyrakenteen pohjaksi käytöspuu, jonka ympärille tehtyyn järjestelmään määriteltiin selkeät kehitysperiaatteet. Järjestelmän jokaisen osan täytyi olla täysin räätälöitävissä, mutta samalla nämä säätömahdollisuudet piti pitää tiukasti hallinnassa. Tekoälyjärjestelmän piti myös pysyä riittävän helppokäyttöisenä pelin tekoälyjärjestelmää käyttäville pelisuunnittelijoille. (Isla 2005.)

Yksi keskeisiä tapoja käytöspuujärjestelmän pitämisessä täysin säädettävänä oli suora puun rakenteen muokkaus ajon aikana. Tämän aikaansaamiseksi Halo 2:ssa käytettiin useita tekniikoita. Ensinnäkin, hahmon käyttäytymismalleille voitiin esimerkiksi määrittää merkkejä (tag), ja vain hahmon tilaan sopivilla merkeillä merkityt mallit otettiin huomioon käyttäytymistä valittaessa. Näin jopa suuria osia koko käytöspuusta saatettiin lukita pois hahmon käytöstä tarpeen sitä vaatiessa. Toisena tapana rakenteen muuttamiselle olivat virikkeet (stimulus), joita voitiin lisätä puuhun tilapäisiksi jäseniksi ja poistaa siitä, kun ne eivät enää olleet tilanteelle tarpeellisia. Viimeiseksi, koska pelistä löytyi paljon hahmotyyppikohtaisia tarpeita, tehtiin järjestelmään mahdollisuus lisätä puuhun hahmokohtaisia käytösmalleja. Näillä puuta luodessa lisätyillä käytösmalleilla luotiinkin suurimmat hahmoille persoonallisuutta luovat ominaisuudet. (Isla 2005.)

Monimutkaiseksi muodostuvan järjestelmän hallinta vaati myös omat ratkaisunsa. Halo 2:n käytöspuussa käytösmallit suoritetaan esimerkiksi aina niiden omalla määritetyllä paikalla, vaikka käytösmallia kutsuttaisiin jostain toisesta puun osasta. Käytännössä puussa siis siirryttiin osasta toiseen. Samoin edellä mainitut virikkeet lisättiin aina itse puuhun, eikä käytösmalleja ikinä suoritettu puun ulkopuolelta. Näillä toimilla voitiin olla varmoja, että hahmon käyttäytymismallit noudattivat haluttua järjestystä, ja että kaikki oli varmasti otettu huomioon.

Toisena itse järjestelmän käytön monimutkaisuutta hallitsevana periaatteena oli laajentaa valmiiden, toimivien ratkaisujen päälle. Tämä näkyi erityisen hyvin tekoälyn toimintaa ohjaavien pelisuunnittelijoiden työkaluissa, joissa eri hahmoille ja niiden tekoälyosille tehtiin oma hierarkiansa. Hahmot saattoivat siten perustua johonkin pohjalla olevaan hahmoon, ja niille tarvitsi määritellä vain pohjasta poikkeavat ominaisuudet uusien hahmojen luomiseksi. (Isla 2005.)

4.3.2 Spore

Maxisin kehittämä Spore on laaja elämän evoluutiota simuloiva peli, jossa pelaaja vie luomansa olion lukuisien eri vaiheiden läpi aina alkulimassa uiskentelusta kaukaisten galaksien valloitukseen asti. Peli sisältää miltei loputtoman määrän erilaisia olentoja, jotka voivat vuorovaikuttaa keskenään tai pelaajan kanssa. (Ocampo 2008.) Olentojen tekoälyjärjestelmän malliksi otettiin edellämainittu Halo 2:n käytöspuumalli, ja pyrkimyksenä oli tehdä siitä paranneltu versio osittain yhteistyössä Halo 2:n tekoälyn kehittäjien kanssa (Hecker 2009).

Sporen ja Halo 2:n tekoälyjärjestelmillä on paljon yhteistä, mutta suurimpana erona Sporen järjestelmässä erotellaan kaksi järjestelmän osaa omiin luokkiinsa: päättäjät ja käytökset. Halo 2:n järjestelmässä käytännössä kaikki puun osat olivat käytöksiä ja käytöksien tärkeysjärjestystä muokkaavia impulsseja. Sporessa toimintaa kuvaavat käytökset eroteltiin päättäjistä, ja näin käytöksiä voidaan käyttää paremmin uudelleen puun eri osassa, eikä Halo 2:n impulssien kaltaisia rakenteita tarvita. (Hecker 2009).

Sporen tekoälyjärjestelmän pohjalle valittiin useita periaatteita, joita se noudattaa kaikissa osissaan. Käytöspuun tilojen täytyy esimerkiksi olla selkeästi ja jämäkästi määritelty niin, että puun yleinen rakenne on helppo hahmottaa. Käytännössä käytöspuun rakenne on siis yhdessä kooditiedostossa, ja varsinainen hahmojen käyttäytyminen voi olla hajautettuna useisiin tiedostoihin. Toisena periaatteena kaikilla käytöspuun osilla täytyy olla jokin oletustoiminnallisuus. Näin järjestelmää käytettäessä uutta koodia täytyy kirjoittaa vain, kun siihen halutaan lisätä jokin ominaisuus, ja kaikki erikseen muokkaamattomat osat käyttävät määriteltyjä oletusarvoja. Kolmantena, käytöspuussa olevien käytöksien tulee välttää lukuisien toisten käytöksien tarkkailua yksittäisissä tapauksessa, etteivät käytöksien väliset riippuvuussuhteet ala kasvaa eksponentiaalisesti. Tarkkailun hoitavat käytöspuun päättäjät, ja käytöksien tuleekin aktivoitua juuri päättäjien hierarkian määrittelemällä tavalla. (Hecker 2009).

5 KÄYTÖSPUUJÄRJESTELMÄN OHJELMOINNIN SUUNNITTELU

Käytöspuita ohjelmoitaessa on hyvä seurata selkeitä periaatteita. Tässä kappaleessa esitellään osat, joiden avulla voidaan toteuttaa selkeä ja toimiva käytöspuujärjestelmä. Kappaleessa käsitellään käytöspuujärjestelmän perustan toteuttamista, siihen liittyviä tukirakenteita, sekä perusosia soveltavia rakenteita. Lisäksi kappaleessa käydään läpi tekoälyn kehittämisessä auttavia työkaluja, kuten erilaisia säätö- ja seurantajärjestelmiä.

5.1 Käytöspuuarkkitehtuurin perusta

Suurien tietomäärien hallitsemiseksi käytöspuujärjestelmän ohjelmoinnissa on syytä rakentaa selkeä joukko perusosia, joista mikä tahansa haluttu toiminnallisuus voidaan kasata. Hyvin määritellyt osat tekevät järjestelmästä helposti laajennettavan ja muokattavan. (Champandard 2008, 257.) Kuten aikaisemmin on mainittu, mahdollisimman yksityiskohtaisesti muokattava järjestelmä on erityisen tärkeää taistelupelien kaltaisissa suunnittelijalähtöisissä peleissä.

5.1.1 Tehtävät

Käytöspuujärjestelmän yksinkertaisimpina osina voidaan käyttää tehtäviä (task), jotka nimensä mukaan suorittavat jonkin toiminnallisuuden halutussa yhteydessä. Tehtävällä ei tarkoiteta mitään tiettyä tekoälyhahmojen käyttäytymistä, vaan sen sijaan eri käyttäytymisvaihtoehdot voidaan rakentaa yhdistelemällä erilaisia tehtäviä. Tehtävät ovat se käytöspuujärjestelmän osa, joka toteuttaa tekoälyn vaatiman toiminnallisuuden konkreettisesti koodissa. (Champandard 2008, 258.)

Käytännössä tehtävät toimivat siis yleisluontoisina palasina, joilla järjestelmän kaikki päätös- ja hallintajärjestelmät voidaan tehdä. Jos tekoälyhahmon täytyy tehdä jokin pelimaailmaan vaikuttava toiminto, voidaan käyttää tehtävää, joka muokkaa hahmoa tai pelimaailmaa suorituksensa aikana. Jos taas tekoälyn täytyy päättää eri valinnoista, voi käytetty tehtävä vain tarkistaa jotakin pelimaailman tilasta, eikä sen välttämättä tarvitse pitää mitään muistissaan. (Champandard 2008, 258.)

Aikaisemmin mainitut tehtävät keskittyvät toimintoihin, jotka suoritetaan välittömästi tai suhteellisen pienen ajan kuluessa. Tätä määritelmää voidaan laajentaa tehtävillä, joita suoritetaan pelimoottorissa useiden päivityskertojen - tai päivityskehysten (update frame) - aikana, ja joiden suoritus voidaan keskeyttää milloin tahansa. Suoritusta jatkettaessa tehtävä on samassa tilassa, kuin mihin se keskeytyessään jäi. Tätä kutsutaan tehtävien jatkuvaksi suorittamiseksi (latent execution). (Champandard 2008, 258.)

5.1.2 Tehtävien lopetustila

Käytöspuujärjestelmän on keskeisesti kyettävä tunnistamaan suoritettavien tehtävien tiloja - erityisesti sen kannalta, kuinka tehtävät saatetaan päätökseen. Tällainen lopetustila (termination status) voidaan tiivistää yhdeksi arvoksi, joka kertoo miten tehtävä päättyi. Käytännössä tehtävän suoritus voi loppua yhdellä tavalla kolmesta: onnistuneesti, odotusten mukaisesti epäonnistuneesti tai odottamattoman virheen myötä. Onnistunut tehtävä on suoritettuaan tehnyt kaiken halutun ja muokannut maailmaa tai tehnyt päätöksensä määritellyllä tavalla. Odotusten mukaisesti epäonnistunut tehtävä ei sen sijaan palautuessaan onnistunut halutussa toimenpiteessä, mutta kaikki suorituksen aikana heränneet ongelmat on otettu huomioon. Kaikkein arvaamattomin tapaus, odottamattomaan virheeseen päättynyt tehtävä, ei saanut toimintaansa halutusti päätökseen, ja muutti mahdollisesti maailman tilaa ennalta määrittelemättömällä tavalla. (Champandard 2008, 258 - 259.)

Järjestelmän täytyy siis toteuttaa jokin tapa, joka tunnistaa kaikki kolme eri tilaa, että tilat voitaisiin käsitellä älykkäästi. Yksinkertaisin ja joustavin tapa on tehdä tehtäviin yksi ainut reitti, jonka kautta niitä käsitellään - käytännössä esimerkiksi yksi kutsuttava funktio tehtävää kuvaavan luokan sisällä - ja palauttaa reitin kautta arvo, joka sisältää kaikki kolme mahdollisuutta. Toisaalta voidaan yksittäisiä tehtäviä toteuttaessa päättää, että tehtävien täytyy aina epäonnistua odotusten mukaisesti. Viimeisenä vaihtoehtona tehtävien suorittamisesta voidaan tehdä kaksivaiheista erottamalla päätös suorituksen aloittamisesta ja varsinainen suoritus omiksi osikseen, sillä vain varsinaisen suorituksen aikana voidaan törmätä odottamattomiin ongelmiin. (Champandard 2008, 259.)

5.2 Perusosien tehokas käyttö

Tekoälyjärjestelmän perusosien käyttöä varten kannattaa rakentaa tukijärjestelmä, jonka avulla osia voidaan käyttää mahdollisimman tehokkaasti. Tukijärjestelmä varmistaa, että tehtävien suoritusta pystytään hallitsemaan ja seuraamaan ja saadut tiedot välittämään eteenpäin niille osille, joissa niitä tarvitaan. (Champandard 2008, 259 - 261.)

5.2.1 Tehtävien hallinta

On usein hyödyllistä keskittää tehtävien suoritus yhteen paikkaan. Kun suoritus on keskitetty, voidaan tehtäviä seurata helposti luotua kanavaa pitkin. Jos tehtäviä yritetään suorittaa liikaa kerralla, voi keskitetty järjestelmä käsitellä ongelmat yksinkertaisemmin kuin eri puolille koodia hajautetussa suorituksessa olisi mahdollista. Luonnollisesti suoritus voidaan myös pysäyttää tai ajoittaa tapahtumaan pidemmällä aikavälillä. Lisäksi tehtävät omistavien tekoälyhahmojen ei tarvitse huolehtia tehtävien suorittamisesta itse, mikäli suoritus on keskitetty. (Champandard 2008, 259.)

Keskittämiseen voidaan käyttää aikatauluttajaa (scheduler), johon tallennettaan kullakin hetkellä suoritettavat tehtävät, ja joka huolehtii tehtävien läpikäymisestä. Tekoälyhahmot pyytävät aikatauluttajaa suorittamaan halutut tehtävät, ja aikatauluttaja lisää ne sisällään olevaan päivitettävien tehtävien listaan. Yksinkertaisimmillaan aikatauluttajan ohjelmoiminen voi olla hyvin suoraviivaista, sisältäen vain tehtävien suoritukseen ja suorituksen pysäyttämiseen liittyvät käyttöliittymät, mutta uusien ominaisuuksien - kuten tehtävien järjestämisen tärkeysjärjestykseen - lisääminen voi tehdä siitä hyvinkin monimutkaisen. (Champandard 2008, 260.)

5.2.2 Tehtävien suorituksen tarkkailu

Aikatauluttajan monimutkaistumista voidaan rajata tekemällä uusista toiminnoista erillisiä komponentteja. Monet laajennukset, kuten erilaisten lokimerkintöjen tekeminen tai muistinhallinnan järjestelmät, voidaan toteuttaa lisäämällä aikatauluttajaan tehtävien suorituksen valmistumista tutkivia yleisiä tarkkailijoita (global observer). Yleisellä tasolla tarkkailijalla tarkoitetaan osaa, joka liittää jonkin tietyn tapahtuman tarkkailtavassa koodissa erillään olevaan koodiin. Yleinen tarkkailija tarkkailee siis aikatauluttajan avulla tehtävien valmistumista, jolloin tapahtumaan voidaan reagoida halutusti. (Champandard 2008, 260.)

Tarkkailijaperiaatetta voidaan soveltaa myös silloin, kun tehtävien täytyy pystyä tarkkailemaan toisia tehtäviä. Tekoälyhahmolle luotavat käytösmallit ovat erityisesti kiinnostuneita juurikin tietyistä tehtävistä, eivätkä niinkään kaikkien tehtävien tarkkailusta. Tarvittava toiminnallisuus saadaan toteuttamalla tehtävien tarkkailija (task observer). Tehtävien valmistumisen - ja etenkin valmistumiseen liittyvän lopetustilan - tarkkailu on myös välttämätön ominaisuus, mikäli käytöspuujärjestelmän toiminta halutaan tehdä hierarkkiseksi. (Champandard 2008, 261.)

Tehtävien tarkkailija on itsekin tehtävä, mutta sisältää käyttöliittymän sille, että sitä voidaan huomauttaa tarkkailtavan tehtävän valmistumisesta ja tulokseksi saadusta lopetustilasta. Tarkkailijan lisääminen suoritettavan tehtävän yhteyteen voidaan toteuttaa esimerkiksi aikatauluttajaan. Kun tehtävä suoritetaan aikatauluttajan kautta, liitetään tehtävään haluttu tarkkailija. Aikatauluttajaa tulee siten myös laajentaa ilmoittamaan tarkkailijalle tehtävän valmistumisesta. (Champandard 2008, 261.)

5.3 Hierarkinen järjestelmä yhdistelmätehtävillä

Käytöspuujärjestelmän logiikan tekeminen hierarkiseksi on erityisen hyvä lähestymistapa tekoälyhahmojen varsinaisten käyttäytymismallien luomiseen. Hierarkisia rakenteita voidaan tehdä kuitenkin useilla tavoilla, ja oikean tavan valitsemisella vältetään useita ongelmia. Kuten luvussa 3.2 jo todettiin, skriptauskielien käyttö on hyvin yleistä taistelupelien tekoälyissä. Skriptausjärjestelmät ovatkin suosittu tapa rakentaa hierarkisuutta, esimerkiksi käyttämällä skriptifunktioita käytöspuun tehtävinä. Tällaisista skriptifunktioista tulee kuitenkin helposti sidonnaisia yksittäisiin tilanteisiin, työläitä kirjoittaa jokaiseen tarvittavaan tilanteeseen ja hankalia, mikäli järjestelmässä tarvitaan useiden käyttäytymismallien yhtäaikaista suoritusta. (Champandard 2008, 261.)

Skriptifunktioita käyttämällä käytösmallit tehdään käytännössä kyseisen skriptauskielen ominaisuuksilla. Edellämainittujen ongelmien välttämiseksi onkin soveliaampaa siirtyä käyttämään valmiiksi rakennettujen komponenttien avulla tehtyjä hierarkioita luomalla tehtävistä parametreillä säädettäviä yhdistelmätehtäviä (composite task). Yhdistelmätehtävät hallitsevat tehtävien joukkoa lapsitehtävinään, ja ohjaavat niitä saatujen syötteiden mukaan. Näiden yhdistelmätehtävien avulla voidaan käytöspuuhun luoda monimutkaisia hallintamalleja käyttämällä aikatauluttajaa ja tehtävien tarkkailijoita. (Champandard 2008, 262.)

5.3.1 Tehtävien peräkkäinen suoritus

Suoraviivaisin tapa toimintojen läpikäymiseen on suorittaa tehtäviä järjestelmällisesti yksi toisensa jälkeen tehtäväsarjana (sequence). Tehtäväsarjat pitävät kirjaa siitä, mitä tehtävää milloinkin ollaan suorittamassa, ja käyttävät aikatauluttajaa itse suoritukseen. Tehtävän valmistumisen seuraamiseen tehtäväsarjat käyttävät tehtävien tarkkailijaa, ja käsiteltyään saadun lopetustilan siirtyvät suorittamaan sarjan seuraavaa tehtävää. (Champandard 2008, 262.)

Kuten kaikkiin yhdistelmätehtäviin, tehtäväsarjoihin saadaan säätömahdollisuuksia lisäämällä niihin parametrejä. Tehtäväsarjat voidaan esimerkiksi määrittää suorittamaan lapsinaan olevat tehtävät vain kerran, tai palaamaan takaisin alkuun kun sarjan loppu on saavutettu. Voidaan myös päättää lopetetaanko koko sarjan suoritus silloin, kun yksi sarjan tehtävistä päättyy epäonnistuneeseen suoritukseen. Monimutkaisempana esimerkkinä tehtäväsarjasta voidaan

tehdä jono, johon voidaan lisätä tai josta voidaan poistaa ajon aikana tehtäviä. (Champandard 2008, 262.)

5.3.2 Tehtävien yhtäaikainen suoritus

Huomattavasti monimutkaisimpia käytösrakenteita saadaan aikaan, kun tehtäviä suoritetaankin useita yhtä aikaa. Tällainen toiminnallisuus voidaan lisätä rinnakkaisena tehtäväsarjana (parallel). Rinnakkaisia tehtäväsarjoja voidaan muokata pääosin määrittämällä kuinka sarja käyttäytyy erilaisten lopetustilojen kanssa. Sarjaan voidaan esimerkiksi määrittää kuinka monen lapsina olevista tehtävistä täytyy onnistua tai epäonnistua ennen kuin itse sarja tuodaan päätökseen. (Champandard 2008, 263.)

Rinnakkaisia tehtäväsarjoja tulee käyttää varovasti, sillä ne tekevät järjestelmän hallinnasta helposti hankalan. Erityisen hyvin ne toimivat kuitenkin kun käsitellään erilaisten ehtojen yhdistelmiä. Jos halutaan esimerkiksi, että suojautuva tekoäly ei lopeta suojautumista niin kauan kuin sitä kohti ammutaan, voidaan suojautumisen aiheuttava tehtävä yhdistää kohti tulevaa tulta tarkkailevan ehdon kanssa. Tällaiset yksittäiset, yhteen paikkaan käytöspuussa sijoittuvat yhdistelmät ovat pääasiassa hyvin turvallisia käyttää. (Champandard 2008, 263.)

5.3.3 Ehtojen käyttö tehtävien suorituksessa

Kuten edellä käsitellystä esimerkistä käy ilmi, erilaiset ehdot tehtävien suoritukseen ovat välttämättömiä tekoälyjärjestelmälle. Päätösten tekemiseen voidaan käyttää valitsijoita (selector). Valitsijat tekevät päätöksen siitä, mitä niiden alla olevista tehtävistä milläkin hetkellä suoritetaan, ja mukauttavat päätöksiään sen mukaan, missä tilassa suoritettava tehtävä palautuu. Käytännössä valitsijat ja tehtäväsarjat luovat koko käytöspuujärjestelmän perustan, ja pystyvät suorittamaan kaikki toiminnot, joihin tekoälyjärjestelmissä on usein käytetty tilakoneita. (Champandard 2008, 262.)

Valitsijoilla on tyypillisesti lukuisia vaihtoehtoja sille, kuinka ne tekevät päätöksensä milloinkin suoritettavasta tehtävästä. Tehtävän valitsemisessa voidaan käyttää todennäköisyyksiä, selkeää arvojärjestystä tai jotain täysin räätälöityä järjestelmää. Valitsijat voidaan myös määri-

tellä tekemään valintansa vain kerran, tai ne voivat vastavuoroisesti tarkkailla päätöstään jatkuvasti ja siirtyä sujuvasti toiseen valintaan. (Champandard 2008, 262.)

5.3.4 Yksittäisten toimintojen laajentaminen

Koska tehtäväsarjat, rinnakkaiset tehtäväsarjat ja valitsijat ovat käytöspuujärjestelmän perusosia, halutaan ne mielellään pitää mahdollisimman yksinkertaisina. Yksittäisten käytösmallien vaatimat ominaisuudet ja toiminnallisuudet voidaankin toteuttaa tehtäviä täydentävinä koristelijoina (decorator). Koristelija on poikkeuksellinen yhdistelmätehtävä, sillä sillä on vain yksi lapsi, ja ne täydentävät tämän lapsen toiminnallisuutta. Ne voivat esimerkiksi rajoittaa käytösmallien suoritusta tietyissä tilanteissa tai pakottaa tehtävälle jonkin tietyn lopetustilan. Koska koristelijat ovat luonnoltaan hyvin uudelleenkäytettäviä, kannattaa suurin osa käytöspuujärjestelmän ominaisuuksista usein koota niiden avulla. Tällöin ratkaisu myös pysyy helposti datapainotteisena. (Champandard 2008, 263.)

5.4 Arvojen säätäminen ja seurantatyökalut

Kuten jo aikaisemmin todettiin, taistelupeleissä pelisuunnittelijoiden täytyy pystyä hallitsemaan erilaisia arvoja erityisen yksityiskohtaisesti. Suunnittelijoilla ei kuitenkaan yleensä ole yhtä hyviä teknisiä taitoja kuin ohjelmoijilla, jolloin valmiin tekoälyn ominaisuuksia ei välttämättä osata käyttää. Tämän takia tekoälyn hallitsemiseen kannattaakin tehdä omat mahdollisimman selkeästi ja helposti käytettävät työkalunsa, joilla pelisuunnittelijat voivat hioa tekoälyn toiminnan mahdollisimman täydelliseksi. (Pfeifer 2008, 27.) Toisaalta tekoäly voi arvoja säädettäessä myös tehdä jotakin täysin arvaamatonta, jolloin tarvitaan täysin erilainen joukko työkaluja. Tekoälyn toimintaa ja toiminnassa ilmeneviä ongelmia voidaan helposti ratkaista rakentamalla tarkoitukseen sopivat seuranta- ja virheenkorjaustyökalut. (Tozour 2002 b, 39.) Tässä kappaleessa esitellään minkälaisia asioita kummankin tyyppisiä työkaluja luodessa kannattaa ottaa huomioon, ja mitä työkaluilla voidaan saada aikaan.

5.4.1 Arvojen säätäminen

Kun tavoitteena on tekoälyn hiominen mahdollisimman toimivaksi, yksi ensimmäisiä määriteltäviä asioita on se, kuinka tarkkaan tekoälyn parissa työskentelevät pelisuunnittelijat haluavat hallita tekoälyn toimintaa. Lähtökohtaisesti kannattaa usein olettaa, että suunnittelijat pystyvät tarvittaessa käyttämään mitä tahansa toimintoja, joita tekoäly itse osaa käyttää. Kaikkien toimintojen käyttämisen ei tarvitse kuitenkaan olla yhtä helppoa, mikäli yksinkertaisten työkalujen tekeminen on liian hankalaa. Käytännössä harvoin tarvittavien ominaisuuksien käyttö voidaan jättää työläämmäksi, mutta jatkuvasti tarvittavien toimintojen tulee olla helposti ulottuvilla. (Pfeifer 2008, 27 - 28.)

Toinen huomioon otettava asia on työskentelyn tekeminen mahdollisimman sujuvaksi. Erilaisissa työkaluissa kannattaa esimerkiksi pitää mahdollisimman yhtenäisiä käytäntöjä, sillä uusien järjestelmien käytön opettelu vie ylimääräistä aikaa ja tekee työskentelystä hankalampaa. Paras ratkaisu onkin usein tehdä pieniä laajennuksia vanhoihin, olemassa oleviin työkaluihin uusia tarvittavia toiminnallisuuksia varten. On myös hyvä ottaa huomioon työkalujen sopivuus niitä käyttäville pelisuunnittelijoille. Paljon teknisiä taitoja omaaville suunnittelijoille voidaan tehdä tehokkaita ja joustavia työkaluja, kuten skriptausta vaativia järjestelmiä, mutta vastaava ratkaisu saattaa olla täysin toimimaton ohjelmointia yhtään osaamattomalle suunnittelijalle. Viimeiseksi, työkaluilla muokatun tekoälyn testaaminen tulee tehdä helpoksi ja nopeaksi. Tekoälyä tulee voida esimerkiksi muokata niin, ettei peliä tarvitse käynnistää uudestaan muutosten testaamiseksi. Näin suunnittelijoiden työaika keskittyy huomattavasti enemmän tekoälyn toiminnan parantamiseen kuin turhaan odotteluun. (Pfeifer 2008, 28 - 29.)

5.4.2 Seuranta ja virheenkorjaustyökalut

Tekoälyn arvojen muokkaukseen tehtyjen työkalujen lisäksi kehitystyössä tarvitaan keinoja, joiden avulla tekoälyn toimintaa voidaan seurata ajon aikana ja havaita siinä olevia virheitä. Erityisen hyviä tähän tarkoitukseen - etenkin pelisuunnittelijoiden kannalta - ovat erilaiset visuaaliset seurantatyökalut, jotka näyttävät pelin ja tekoälyn tiloja ja muuta testauksen kannalta tärkeää tietoa havainnollisessa muodossa, kuten viivoina, ympyröinä, nuolina tai tekstinä. Tällaisella tiedolla voidaan esimerkiksi varmistaa, että tekoäly toimii ennakoidulla tavalla, saada kiinni ohjelmointiympäristön virheenetsintätyökaluilla (debugger) vaikeasti ajoitettavia ongelmia, ja havaita virheitä, joissa tekoälyhahmot vaihtelevat tilojen välillä väärällä tavalla. (Schwab 2004, 560 - 561.)

Näytettävää tietoa voidaan ryhmitellä sopiviin kokonaisuuksiin, joiden avulla voidaan tutkia jotain tekoälyn toiminnan tiettyä puolta. Kaikkein yleisluontoisin näistä on tekoälyhahmojen yleinen tieto. (Pfeifer 2008, 34.) Yleistä tietoa ovat esimerkiksi hahmon tunnistamiseen vaadittavat tiedot, kuten sen nimi tai tunnistukseen käytettävä numero, hahmon tyyppi, sekä mahdolliset pelilliset arvot, kuten terveyspisteet (hit points) tai panssarointi (armor) (Tozour 2002, 42). Lisäksi tähän ryhmään voidaan sisällyttää fysiikkaan liittyviä ominaisuuksia, kuten hahmon sijainti pelimaailmassa ja sen liikenopeus. (Pfeifer 2008, 34.)

Toisena ryhmänä voidaan käsitellä hahmon animaatioihin liittyvät ominaisuudet (Pfeifer 2008, 34). Näytettävää tietoa voivat olla esimerkiksi hahmon katseluhetkellä toistettavat animaatiot, sekä mahdolliset komennot, joita hahmo käyttää komentaakseen pelin animaatiojärjestelmää (Tozour 2002, 43 - 44). Animaatioista voidaan näyttää myös tarkemmin animaation nimi, pituus, ja missä tilassa itse animaatio on (Pfeifer 2008, 34).

Suurena ja erityisen keskeisenä ryhmänä ovat tekoälyn tilaan liittyvät tiedot. Tekoälyn toiminnasta voidaan näyttää sen aikomuksiin, havaintoihin ja nykyiseen sekä edellisiin tiloihin liittyviä asioita. (Pfeifer 2008, 35.) Hyökkäävästä tekoälyhahmosta voidaan esimerkiksi piirtää nuoli sen nykyiseen kohteeseen, tai vastaavasti nuolella voidaan merkitä mitä hahmoa joukossa oleva tekoälyhahmo pitää johtajanaan (Tozour 2002, 43). Tekoälyn aikomuksista voidaan näyttää myös toiminnot, joita tekoäly haluaa suorittaa - huolimatta siitä onnistuuko se aikomuksissaan. Havaintoihin liittyen voidaan sen sijaan näyttää esimerkiksi listoja erilaisista kappaleista, kuten vihollisista, joita tekoäly on lähiaikoina nähnyt tai näkee parhaillaan ja ottaa huomioon päätöksiään tehdessä. (Pfeifer 2008, 35.) Tätä järjestelmää voidaan myös tar-

kentaa lisäämällä mahdollisuuden tutkia erilaisilla aisteilla tapahtuvia havaintoja. Lopuksi, tekoälyn tilaa voidaan seurata näyttämällä hahmojen nykyinen tila, tai jos mahdollista, koko käytetyn tilajärjestelmän tiedot (Tozour 2002, 42).

Tekoälyn tilasta ja aikomuksista erillisinä tietona voidaan pitää reitinhakuun liittyviä tietoja (Pfeifer 2008, 35). Monimutkaisessa tapauksessa voidaan näyttää esimerkiksi mahdolliset etukäteen lasketut pisteet, joita pitkin reitinhaku tapahtuu. Kolmiulotteisissa peleissä tämä tarkoittaa usein reitinhakuverkkoa (navigation mesh), joka näyttää, miten pelimaailman pinnoilla voidaan kulkea. Voidaan myös näyttää yksityiskohtaisesti, minkälaisten päätösten kautta tekoälyhahmo löytää mahdollisen käytetyn reittinsä, itse reitti, jota pitkin hahmo on juuri kulkemassa, ja jopa lista paikoista, joissa tekoäly on viime aikoina liikkunut. (Tozour 2002, 42 - 43.)

Yllämainitun kaltaista tietoa näyttävistä osista kannattaa kaikista tehdä omia, toisistaan täysin riippumattomia palasia (Tozour 2002, 42). Näiden tietoryhmien näyttämisen tulee voida pystyä asettamaan päälle tai pois esimerkiksi näppäinkomennoilla, että tekoälyhahmossa olevia virheitä voidaan tarkkailla eri näkökulmista nopeasti. Näytetty tieto kannattaa myös pitää niin havainnollisena, että sen tarkoituksen voi ymmärtää jo pelkällä vilkaisulla. Tähän päästään käyttämällä esimerkiksi havainnollisia muotoja ja sopivia värejä. Hyvänä esimerkkinä tekoälyhahmosta voidaan piirtää nuoli sijaintiin, johon se yrittää liikkua. Nuolesta voidaan tehdä huomattavasti havainnollisempi, jos se väritetään esimerkiksi vihreäksi, keltaiseksi tai punaiseksi riippuen siitä, onko hahmon reitinhaku onnistunut, vielä kesken vai epäonnistunut. Tiedon näyttävää järjestelmää tulee myös pitää yllä. Aina kun tekoälystä löydetään virhe, on hyvä lisätä järjestelmään sopivaa tietoa näyttävä osa, jolla vastaavat virheet voidaan välittömästi löytää jatkossa, jolloin säästetään huomattavasti virheiden etsintään kuluvaa aikaa. (Pfeifer 2008, 35.)

Tekoälyn toimintojen lisäksi tietoa voidaan näyttää virheellisesti syötetyistä arvoista johtuvista virhetilanteista. Aina kun tällainen virhe tapahtuu, on se hyvä näyttää ruudulla virheilmoituksella. Virheilmoituksen tulee olla mahdollisimman selkeä, että myös ihminen, jolla on vähemmän ohjelmointikokemusta, pystyy ymmärtämään, mistä kohdattu virhe johtuu. Mikäli huomataan, että jokin virheilmoitus on liian epäselvä, kannattaa se heti muokata paremmin tilanteeseen sopivaksi. (Pfeifer 2008, 35.)

6 TEKOÄLY- JA SEURANTAJÄRJESTELMIEN TOTEUTUS

Käytännön projektin tarkoituksena on toteuttaa toimiva ja yleiskäyttöinen käytöspuujärjestelmä, jota voidaan käyttää yksinkertaisen tekoälyn tekemiseen Access Lumia –taistelupeliin. Käytöspuujärjestelmän tueksi on tarkoitus myös tehdä seuranta- ja virheenetsintäjärjestelmä, ja tutkittua seurantajärjestelmän hyötyjä tekoälyn kehityksessä. Kappaleessa määritellään projektin vaatimukset, dokumentoidaan ohjelmoidut osat ja käydään läpi järjestelmien toteutus- ja testausvaiheet.

6.1 Vaatimusmäärittely

Tekoälyhahmon liikkuminen ja hyökkääminen muodostavat taistelupelin perustan. Access Lumia -pelissä kenttä on yksinkertaisesti taso, joten liikkumiselta ei vaadita mitään äärimmäisen monimutkaista. Keskeisesti, tekoälyhahmon täytyy osata liikkua pelaajaa kohti sopivalle hyökkäysetäisyydelle ja tarvittaessa perääntyä, jos tilanne käy vaaralliseksi. Perääntyminen tulee lopettaa, kun etäisyyttä kohteeseen on riittävästi. Oleellisesti liikkuminen tuo siis seuraavat vaatimukset: hahmon tulee osata liikkua kohdettaan kohti ja kohteesta poispäin, hahmon tulee pystyä tunnistamaan, milloin se on tarpeeksi lähellä kohdettaan, ja milloin se on riittävän kaukana kohteesta.

Koska hyökkääminen on toinen taistelupelin perustoiminnallisuuksista, tekoälyhahmon pitää selvästi myös osata hyökätä. Hahmo ei saa kuitenkaan huijata, vaan sen täytyy olla samalla viivalla ihmispelaajan kanssa. Tämän takia tekoälyhahmon tulee käyttää samoja näppäinkomentoja hyökkäyksiensä tekemiseen kuin pelaajakin. Tekoälyn tulee siis osata tuottaa ulos syötettä, joiden avulla hyökkäyksiä tehdään. Samaa periaatetta tulee myös soveltaa yllämainitussa liikkumisessa.

Kuten luvussa 3 todettiin, liikesarjat ovat nykyään hyvin keskeisessä roolissa taistelupeleissä. Tekoälyn tulee siis osata yhdistellä liikkeistä liikesarjoja. Tekoälyn täytyy osata suorittaa näppäinkomentoja oikealla ajoituksilla peräkkäin niin, että tietystä hyökkäyksestä voidaan tehdä tiettyjä liikesarjoja. Mikäli liikesarja katkeaa, esimerkiksi sen takia, että hahmo ottaa vahinkoa, tekoälyn tulee osata keskeyttää liikesarjan tekeminen aivan kuin yksittäinen sen tekemä liike olisi keskeytetty.

Perääntymiskäyttäytymiseen kuuluu liikkumisen lisäksi myös se, että hahmo tunnistaa, milloin sen täytyy aloittaa perääntyminen. Perääntymishalu tulee toteuttaa pelkotaso-arvolla, jota voidaan kerryttää minkä tahansa halutun toiminnon sisältä, erityisesti silloin, kun tekoälyhahmon elämäpisteet (hit points) vähenevät. Myös pelkotason nollaamisen tulee olla mahdollista. Perääntymistila ei saa myöskään käynnistyä liian usein, koska se tekisi tekoälyn käyttäytymisestä pelaajan kannalta ärsyttävää. Käynnistämiselle tarvitaan siis jonkinlainen rajoittava osa, joka antaa tilan käynnistyä vain kerran tietyllä aikavälillä.

Kuten opinnäytetyössä on jo aiemmin perusteltu, pelin pelaamisen tekeminen hauskaksi ja viihdyttäväksi on usein tekoälyn toiminnan kannalta tärkein ominaisuus. Että tekoäly olisi viihdyttävä, sen tuoman haastavuustason täytyy olla pelaajalle sopiva. Tekoälyn vaikeustason täytyy siis olla säädettävissä. Erityisiä säätökeinoja ovat tekoälyn reaktioaikojen säätö, tekoälyn aggressiivisuus ja tekoälyn kyky käyttää voimakkaita liikesarjoja.

Aikaisemmin on myös mainittu, että tekoälyn toimintojen tulee vaikuttaa perustelluilta ja järkeviltä. Tekoäly ei siis saa jäädä jumiin liikkumisessaan, eikä se saa lukittua tekemään vain yhtä hyökkäystä. Tekoälyn tulee käyttää pelin eri ominaisuuksia, kuten hyppyjä ja ilmahyökkäyksiä, yksittäisiä liikkeitä ja liikesarjoja, vaihtelevasti. Tekoälyn tulee myös rytmittää tekonsa ihmismäisesti. Sen toiminnassa tulee olla hetkiä, jolloin se antaa pelaajalle tilaa juoksemalla pakoon tai odottamalla hyökkäyksien välissä, ja hetkiä, jolloin se hyökkää aktiivisesti pelaajan kimppuun.

Teknisesti itse tekoälyjärjestelmältä vaaditaan ensisijaisesti sitä, että eri tiloja voidaan lisätä ja muokata mahdollisimman helposti. Tekoälyn eri toiminnallisuuksien tulee siis olla hyvin jaoteltu omiin, järkeviin osiinsa. Käytännössä parhaiten tämä saadaan aikaiseksi toteuttamalla luvussa 5 määritellyt osat käytöspuujärjestelmän pohjaksi.

Seurantajärjestelmä tehdään erillisenä osana tueksi tekoälyjärjestelmälle. Seurantajärjestelmän tulee pystyä näyttämään täysin räätälöitävissä olevaa tietoa omissa näkymissään, ja näkymien välillä tulee pystyä siirtymään näppäimiä painamalla. Näin eri näkymillä voidaan etsiä ratkaisua ilmeneviin ongelmiin seuraamalla tekoälyn tai pelihahmojen tilaa mahdollisimman monesta eri näkökulmasta luvun 5.4.2 periaatteiden mukaisesti. Uusia näkymiä tulee voida lisätä järjestelmään helposti.

Opinnäytetyötä varten tulee toteuttaa erityisesti tekoälyn kannalta kolme oleellista näkymää. Ensimmäinen näistä on tekoälyhahmosta perustietoa näyttävä näkymä. Perustietoon kuuluu

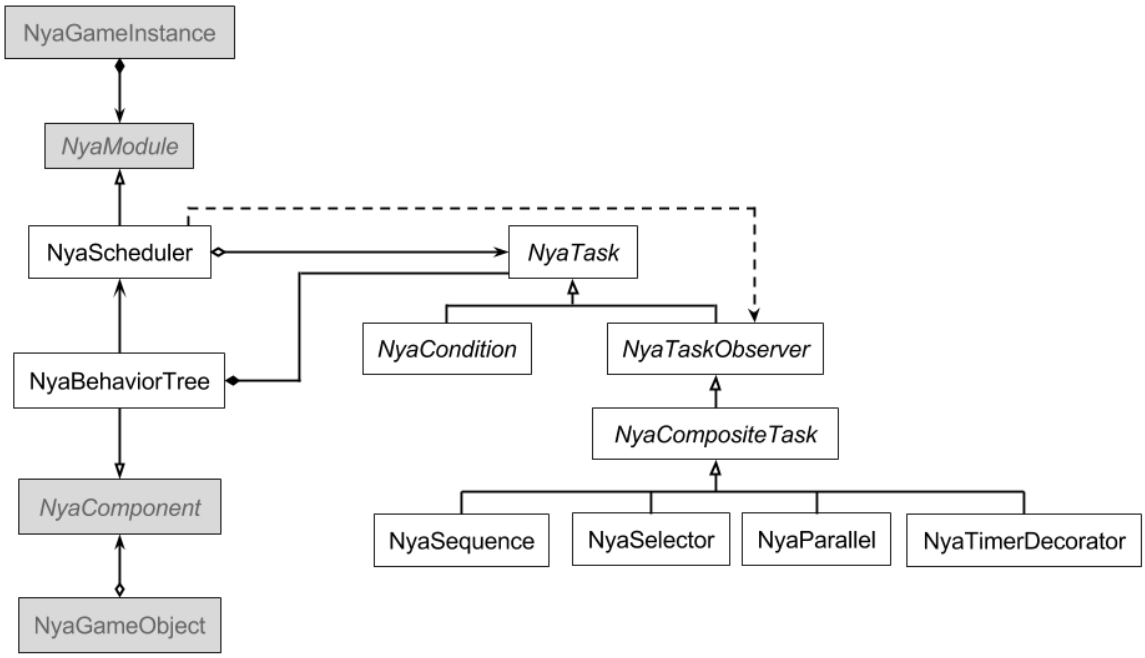
hahmon nimi, sijainti ruudulla, nopeus ja se, missä animaatiotilassa hahmo milläkin hetkellä on. Toisena, tekoälyn kannalta oleellisimpana näkymänä, on tekoälyn tilan näyttävä näkymä. Parhaiten tekoälyn tila saadaan näkyviin piirtämällä reaaliaikainen kuva käytöspuun kaikista jäsenistä ja siinä aktiivisena olevasta reitistä. Viimeisenä tarvitaan hahmojen törmäystarkastusmuodot näyttävä näkymä, jonka avulla voidaan selvittää ongelmia hyökkäys- ja taistelujärjestelmissä. Törmäystarkastusmuodot asetellaan Access Lumia -pelissä erikseen joka animaatiokehykselle, joten mahdollisuus tarkastella niitä reaaliaikaisesti on tärkeää.

6.2 Tekninen suunnitelma

Teknisessä suunnitelmassa puretaan auki opinnäytetyön aikana kehitetyt kolme kokonaisuutta: käytöspuujärjestelmä, Access Lumia -peliin kehitetyt tekoälyosat, sekä tekoälyjärjestelmän tukena oleva seurantajärjestelmä. Kaikista kokonaisuuksista esitellään niiden arkkitehtuuri. Kokonaisuuden luokat on dokumentoitu kunkin kokonaisuuden alle, ja luokista esitetään luokkakaaviot, joista käy ilmi luokkien rakenne, ja kuinka ne liittyvät muihin luokkiin.

6.2.1 Käytöspuujärjestelmä (NyaBehaviorTreeEngine)

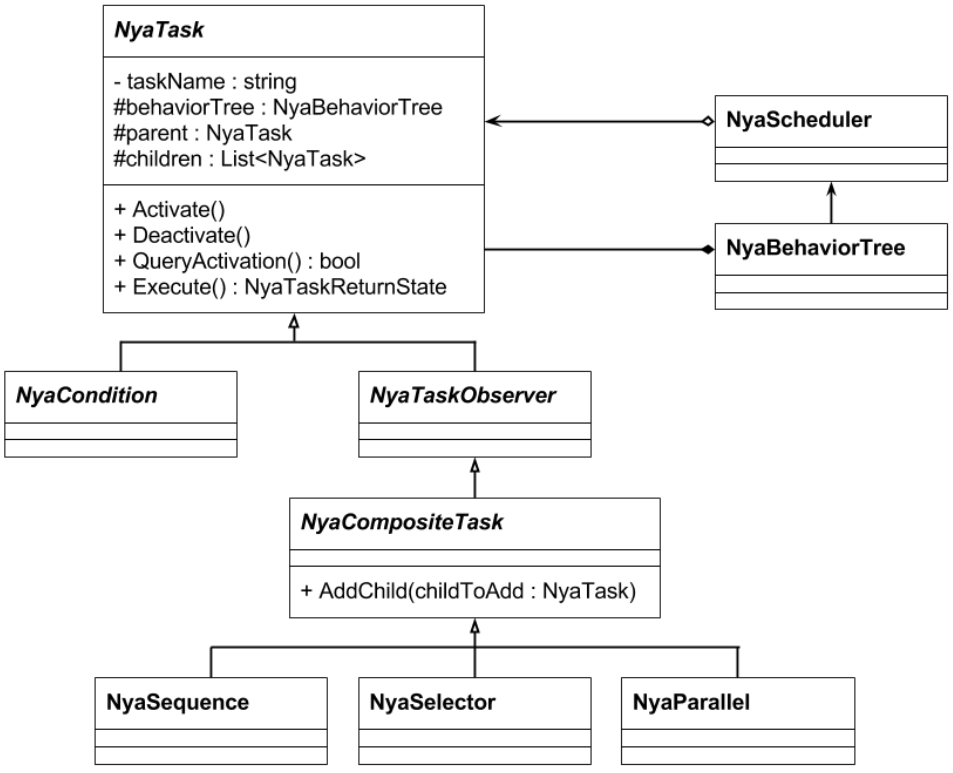
NyaBehaviorTreeEngine on kirjasto, joka mahdollistaa tekoälyä varten tarvittavan käytöspuun rakentamisen. Kirjasto toimii kuitenkin vain tekoälyjärjestelmän perustana, sillä sen avulla voidaan luoda käytöspuun tarvitsema rakenne, käydä käytöspuuta läpi ja suorittaa kirjaston ulkopuolella luotuja toimintoja, mutta kirjasto itsessään ei sisällä mitään valmista toiminnallisuutta. Toiminnallisuudella tarkoitetaan tässä yhteydessä varsinaisia tekoälyn suorittamia tekoja, kuten hahmon liikkumista tai hyökkäämistä. Toiminnallisuus luodaan perimällä käyttötarkoitukseen sopivia luokkia kirjaston abstrakteista komponenteista (Kuvio 4). Kirjastoon kuuluvat osat on toteutettu luvuissa 5.1, 5.2 ja 5.3 esiteltyjen periaatteiden mukaisesti.



Kuvio 4. NyaBehaviorTreeEngine-käytöspuukirjaston arkkitehtuuri. Harmaalla merkityt luokat eivät kuulu kirjastoon. Kursivoitu teksti tarkoittaa abstraktia luokkaa.

NyaTask

NyaTask on abstrakti luokka, joka toimii luvun 5.1.1 mukaisesti käytöspuun yksinkertaisena perusosana, tehtävänä. Käytännössä käytöspuun hierarkian jokainen osa on peritty NyaTask-luokasta, ja luokka toimiikin perustana kaiken toiminnallisuuden toteuttamiseen. Hierarkia rakentuu lisäämällä tehtävälle lapsitehtäviä luokassa olevaan children-muuttujaan (Kuvio 5). Koska NyaTask-luokka itsessään ei osaa kuitenkaan huolehtia lastensa suorittamisesta, se ei sisällä käyttöliittymää lapsitehtävien lisäämiseen. Lapsitehtävien lisääminen tapahtuu sen sijaan myöhemmin tarkemmin käsiteltävän NyaCompositeTask-luokan kautta. Tehtävien suorittamisen kannalta lista lapsitehtävistä oltaisiin voitu toteuttaa myös NyaCompositeTask-luokkaan, mutta NyaTask-luokkaan rakennettuna käytöspuun hierarkiaan päästään paremmin käsiksi järjestelmän ulkopuolelta. Tämä muodostuu erityisen tärkeäksi seurantajärjestelmän kannalta, kun pyritään mallintamaan käytöspuun rakennetta käyttäjälle.



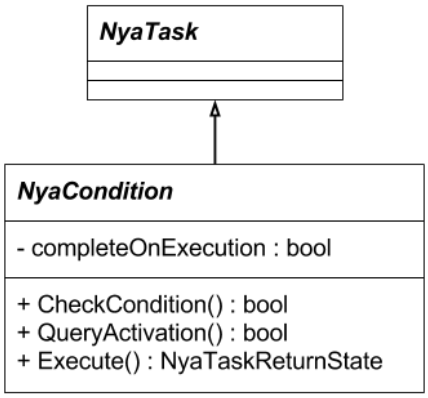
Kuvio 5. NyaTask-luokan luokkakaavio

NyaTask-luokan käyttöliittymässä on kaksi virtuaalista metodia, jonka kautta sitä käytetään. QueryActivation()-metodilla käytöspuu, joka on toteutettu NyaBehaviorTree-luokkaan, kysyy tehtävältä, haluaako se aktivoitua. Mikäli tehtävä haluaa aktivoitua, se yleensä lisätään

suoritettavaksi NyaScheduler-luokasta tehtyyn aikatauluttajaan. Varsinainen suoritettava toiminnallisuus suoritetaan NyaTask-luokan Execute()-metodissa. Execute()-metodi palauttaa jokaisella suorituskerralla NyaTaskReturnState-tyyppisen palautusarvon, joka kertoo missä tilassa tehtävä suorituksen jälkeen on luvun 5.1.2 periaatteiden mukaisesti. Arvo NyaTaskReturnState.Running tarkoittaa, että tehtävä on vielä käynnissä, ja sen suoritusta aiotaan jatkaa. NyaTaskReturnState.Completed sen sijaan palautetaan, kun tehtävän suoritus on saatettu päätökseen onnistuneesti. Mikäli suoritus on jouduttu päättämään keskeneräisenä, palautetaan NyaTaskReturnState.Failed. Kaikki muut tapaukset palauttavat arvon NyaTaskReturnState.Error, joka tarkoittaa tehtävän joutuneen virhetilaan.

NyaCondition

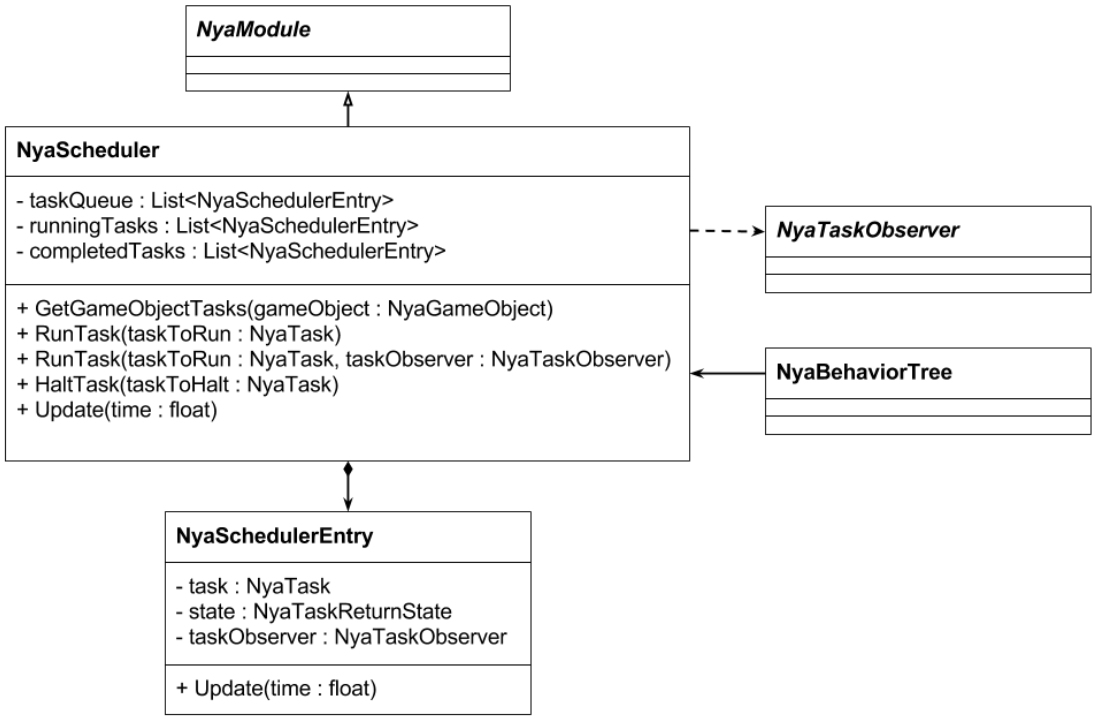
NyaCondition on NyaTask-luokasta peritty abstrakti luokka, joka tarkastelee täyttyykö jokin määritelty ehto. Tämä ehto määritellään toteuttamalla NyaCondition-luokista perityssä luokassa virtuaalinen CheckCondition()-metodi. Totuusarvon (bool) palauttavassa metodissa määritellään se logiikka, jolla halutun ehdon tulee täyttyä. NyaTask-luokasta uudelleen toteutetut QueryActivation() ja Execute() on ohjattu käyttämään CheckCondition()-metodia toiminnassaan. QueryActivation() palauttaakin vain suoraan CheckCondition()-metodin palautusarvon, ja Execute()-metodissa palautetaan joko NyaReturnState.Running, mikäli CheckCondition()-metodin palautusarvo on tosi, tai NyaReturnState.Failed, jos palautusarvo on epätosi. Jos NyaCondition-luokan completeOnExecution-lippu on asetettu, palautetaan NyaReturnState.Running-arvon sijasta NyaReturnState.Completed. (Kuvio 6.)



Kuvio 6. NyaCondition-luokan luokkakaavio

NyaTaskObserver

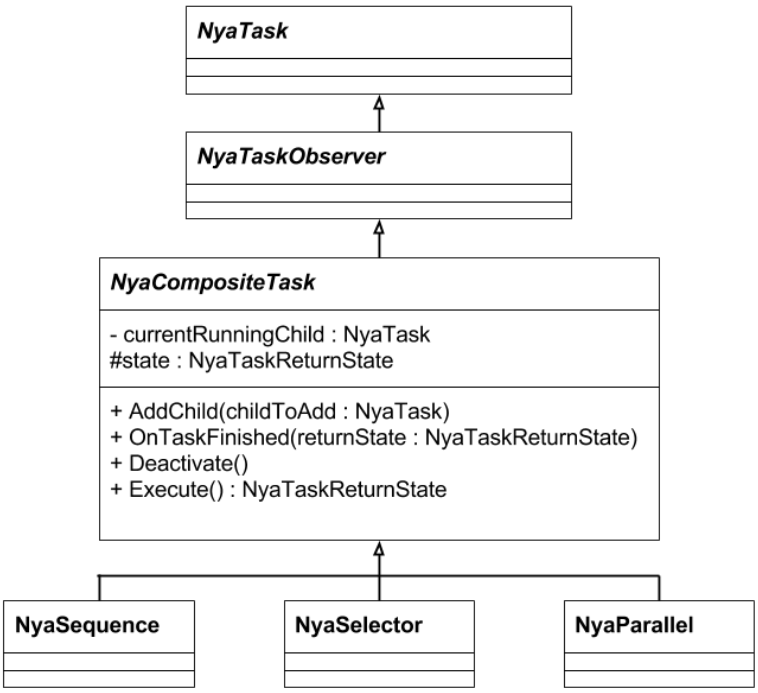
NyaTaskObserver on NyaTask-luokasta peritty abstrakti tehtäväluokka, joka pystyy NyaScheduler-luokan avulla seuraamaan, milloin jonkin tehtävän suoritus päättyy. Kun tehtävä, jonka yhteyteen NyaTaskObserver-olio on liitetty NyaScheduler-luokan RunTask()-metodin kautta, valmistuu, ilmoitetaan siitä NyaTaskObserver-oliolle kutsumalla sen OnTaskFinished()-metodia. NyaTaskObserver-luokka toimii pohjana lapsitehtäviä käsittelevälle NyaCompositeTask-luokalle. Vaikka luokka itsessään on äärimmäisen yksinkertainen, ja sen toiminnallisuus olisi voitu sisällyttää NyaCompositeTask-luokkaan, on se eroteltu omaksi luokakseen, koska tekoälyjärjestelmää laajennettaessa on hyvin mahdollista, että tarvitaan toisia tehtäviä seuraavia tehtäväluokkia, jotka eivät tarvitse NyaCompositeTask-luokan käyttöliittymiä. Tämän opinnäytetyön laajuudessa tällaisia luokkia ei kuitenkaan ole. (Kuvio 7.)



Kuvio 7. NyaTaskObserver-luokan luokkakaavio

NyaCompositeTask

NyaCompositeTask on NyaTaskObserver-luokasta peritty abstrakti tehtäväluokka, joka käsittelee lapsinaan olevia tehtäviä. NyaCompositeTask toimii tekoälyjärjestelmän hierarkisen logiikan pohjana, sillä se mahdollistaa lapsitehtävien lisäämisen tehtäville. Käytännössä käytöspuun haarat siis muodostuvat NyaCompositeTask-luokasta perittyjen luokkien olioista. Luokasta peritään kaikki luvussa 5.3 määritellyt hallintarakenteet, kuten tehtäväsarjat, valitsijat ja koristelijat.

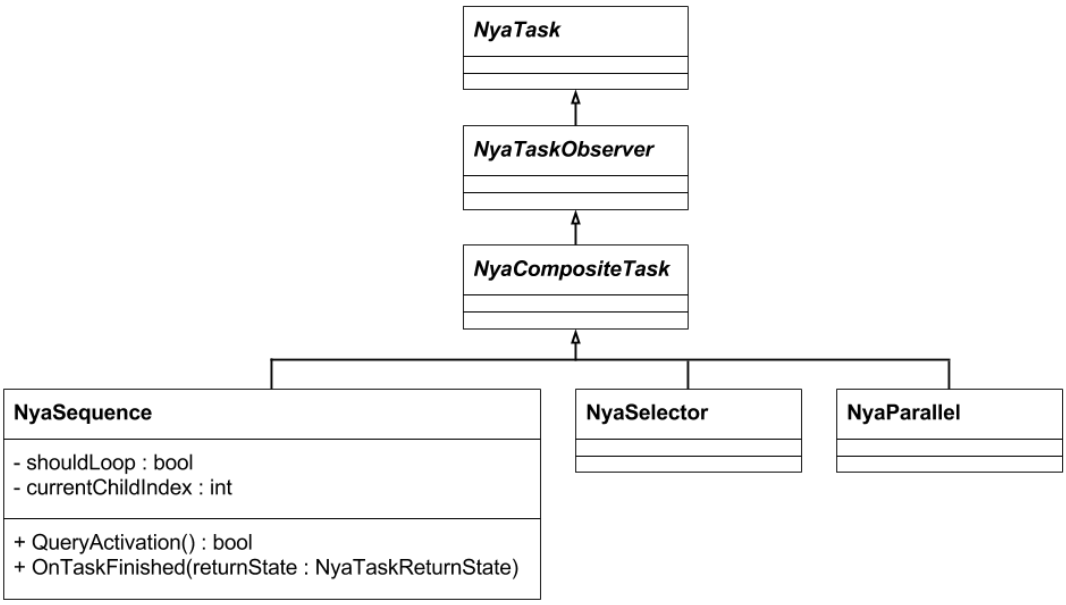


Kuvio 8. NyaCompositeTask-luokan luokkakaavio

NyaCompositeTask-luokasta perityt luokat käyttävät NyaTask-luokan virtuaalisia metodeja yhteisiä periaatteita noudattaen. QueryActivation()-metodi lisää suoritettavan tehtävän - tai NyaParallel-luokan tapauksessa suoritettavat tehtävät - NyaScheduler-olioon sen RunTask()-metodin kautta, ja lisäävät itsensä tehtävän seuraajaksi. OnTaskFinished()-metodissa määritellään, millä tavalla valmistuneita lapsitehtäviä käsitellään, ja miten ne vaikuttavat itse yhdistelmätehtävän tilaan, joka on tallennettu state-muuttujaan. Lopulta, Execute()-metodi palauttaa state-muuttujassa olevan tilan, ja voi siten päättää yhdistelmätehtävän toiminnan. (Kuvio 8.)

NyaSequence

NyaSequence on luvussa 5.3.1 määriteltyä tehtäväsarjaa kuvaava yhdistelmätehtäväluokka, joka on peritty NyaCompositeTask-luokasta. Luokka suorittaa lapsikseen lisättyjä tehtäviä peräkkäin yksi kerrallaan, kunnes koko sarja on suoritettu, tai jokin lapsista lopettaa suorituksen epäonnistuneesti palauttamalla tilan NyaReturnState.Failed. Mikäli totuusarvon sisältävä shouldLoop-lippumuuttuja on asetettu, aloitetaan sarjan suoritus taas ensimmäisestä jäsenestä, kun sen viimeisen jäsenen suoritus valmistuu.



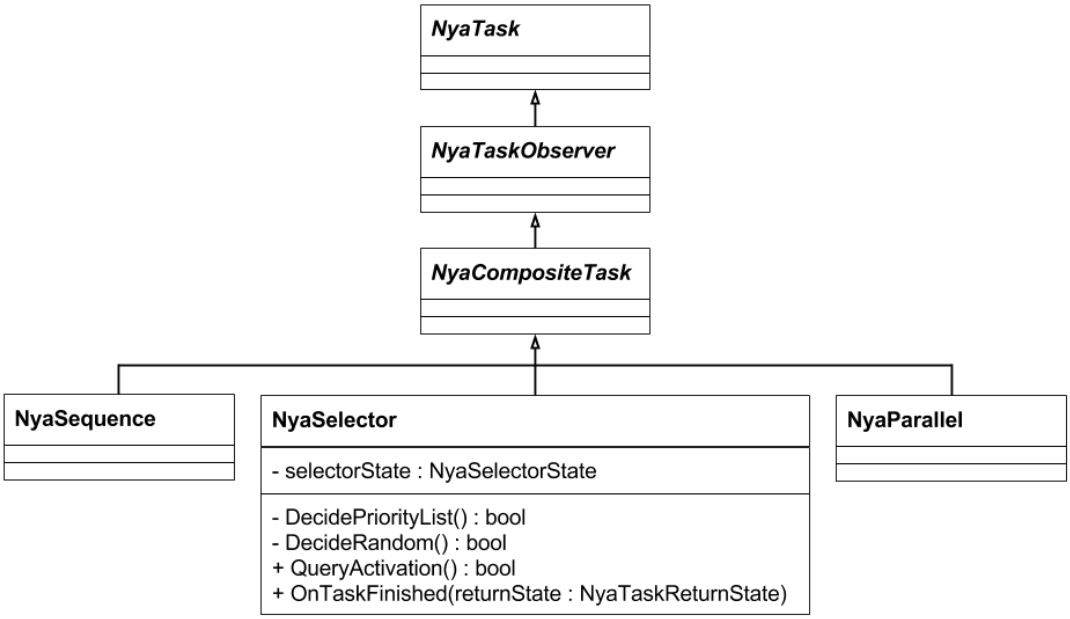
Kuvio 9. NyaSequence-luokan luokkakaavio

NyaSequence tarkistaa QueryActivation()-metodissa haluaako sarjan ensimmäinen lapsitehtävä aktivoitua, ja aktivoituu itse mikäli lapsitehtävän QueryActivation()-metodin palautusarvo on tosi. Aina kun suoritettavana olevan lapsen suoritus päättyy onnistuneesti, valitaan OnTaskFinished()-metodissa seuraava suoritettava lapsi. Luokka käyttää currentChildIndex-muuttujaa pitämään kirjaa siitä, mitä lasta milläkin hetkellä ollaan suorittamassa. (Kuvio 9.)

NyaSelector

NyaSelector on NyaCompositeTask-luokasta peritty luvussa 5.3.3 esitellyn valitsijan toteuttava yhdistelmäluokka. NyaSelector valitsee lapsitehtävistään kulloinkin suoritettavan tehtävän selectorState-muuttujaan asetetun logiikan perusteella. Mikäli muuttujan arvo on NyaSelectorState.PriorityList, suoritetaan lapsia arvojärjestyksessä, tai jos arvo on NyaSelectorState.Random, suoritetaan jokin lapsitehtävistä satunnaisesti.

Tilasta riippuen QueryActivation()-metodi ohjaa kysymyksen tehtävän aktivoitumisesta DecidePriorityList()- tai DecideRandom()-metodille. DecidePriorityList()-metodi valitsee suoritettavaksi lapseksi ensimmäisen lapsitehtävän, joka haluaa aktivoitua. Mikäli mikään tehtävistä ei aktivoidu, ei myöskään itse valitsija aktivoidu. DecideRandom()-metodi sen sijaan kasaa listan kaikista niistä lapsistaan, jotka haluavat aktivoitua, ja valitsee sitten jonkin niistä satunnaisesti. Kuten DecidePriorityList()-metodikin, myös DecideRandom()-metodi palauttaa arvon epätosi, jos mikään sen lapsista ei halua aktivoitua. (Kuvio 10.)



Kuvio 10. NyaSelector-luokan luokkakaavio

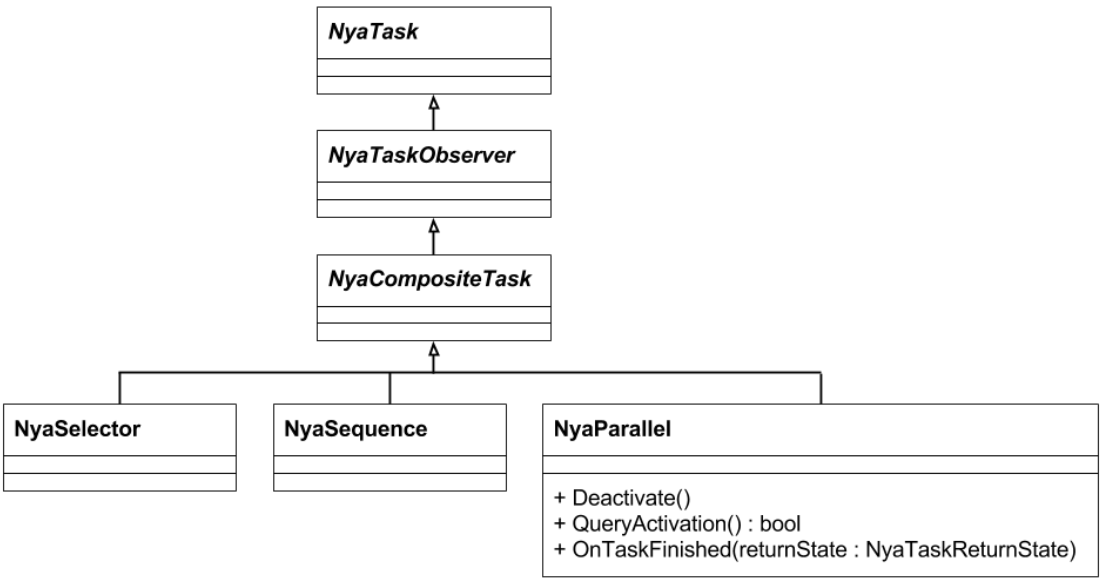
Myös lapsitehtävän suorituksen valmistuminen käsitellään eri tavalla OnTaskFinished()-metodissa, riippuen kummantyyppinen tila valitsijalle on asetettu. Arvojärjestysvalitsija pyytää aina DecidePriorityList()-metodia valitsemaan uuden suoritettavan lapsen. Mikäli suoritettavaa lasta ei löydy, asetetaan valitsijan tilaksi NyaReturnState.Failed, jolloin sen Execu-

te()-metodi palautuu epäonnistuneesti. Satunnainen valitsija sen sijaan palauttaa suoraan lapsitehtävän palautustilan, eikä tee valintaa uudestaan. (Kuvio 10.)

NyaSelector-luokan lapsitehtävien valintalogiikka olisi voitu toteuttaa paremmin erikseen määritellyillä luokkarakenteilla, jolloin valintatyyppejä olisi voinut olla rajattomasti. Yksinkertainen luokkaan itseensä rakennettu tilajärjestelmä oli kuitenkin tämän opinnäytetyön kannalta riittävä ratkaisu.

NyaParallel

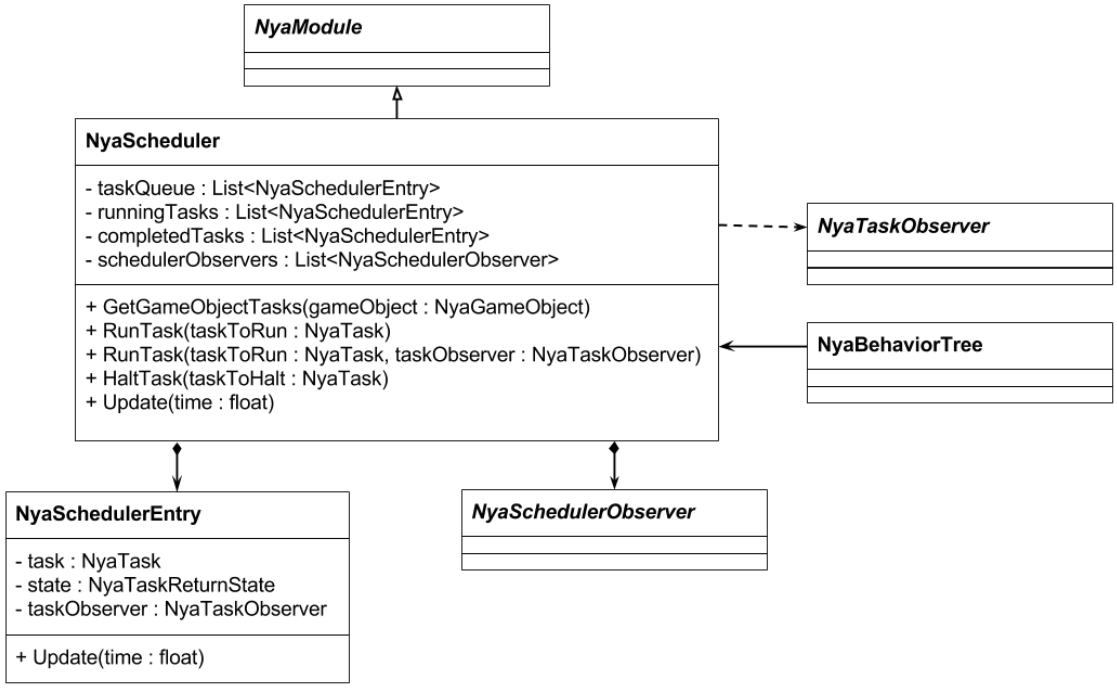
NyaParallel on yhdistelmätehtäväluokka, joka suorittaa lapsinaan olevia tehtäviä yhtä aikaa. Se toimii siis luvun 5.3.2 mukaisena rinnakkaisena tehtäväsarjana. NyaParallel-luokka toimii hyvin pitkälti samalla tavalla kuin NyaSequence-tehtäväsarjaluokka, mutta soveltaa periaatteita yhden tehtävän sijasta kaikkiin sarjaan lisättyihin lapsitehtäviin. QueryActivation()-metodissaan kaikkien lapsitehtävien tulee palauttaa omissa QueryActivation()-metodeissaan arvon tosi, tai sarja ei aktivoidu. Mikäli sarja aktivoituu, se lisää NyaScheduler-olioon suoritettavaksi kaikki lapsitehtävänsä. Samoin, jos yksikin lapsista valmistuu ja palauttaa arvon NyaReturnState.Failed tai NyaReturnState.Completed, kaikkien lapsien suoritus pysäytetään, ja sarjan Execute()-metodissa palautetaan sama palautusarvo. (Kuvio 11.)



Kuvio 11. NyaParallel-luokan luokkakaavio

NyaScheduler

NyaScheduler on tehtäviä keskitetysti suorittava yksinkertainen aikatauluttaja, joka on toteutettu luvun 5.2.1 mukaisesti. Tehtävä lisätään NyaScheduler-luokkaan sen RunTask()-metodin kautta. Metodi luo suoritettavalle tehtävälle NyaSchedulerEntry-tyyppisen olion, johon talletetaan itse tehtävä ja joka pitää kirjaa tehtävän NyaReturnState-tyyppisestä palautustilasta ja siitä, täytyykö jollekin NyaTaskObserver-oliolle ilmoittaa, kun talletettu tehtävä valmistuu. Luotu NyaSchedulerEntry-olio lisätään sitten taskQueue-jonotuslistaan, josta se edelleen siirretään suoritettavana olevat tehtävät sisältävään runningTasks-listaan. Tehtäviä ei voida suoraan lisätä runningTasks-listaan, koska kutsu RunTask()-metodiin on voinut tulla tehtävältä, jota ollaan juuri käyty listassa läpi. (Kuvio 12.)



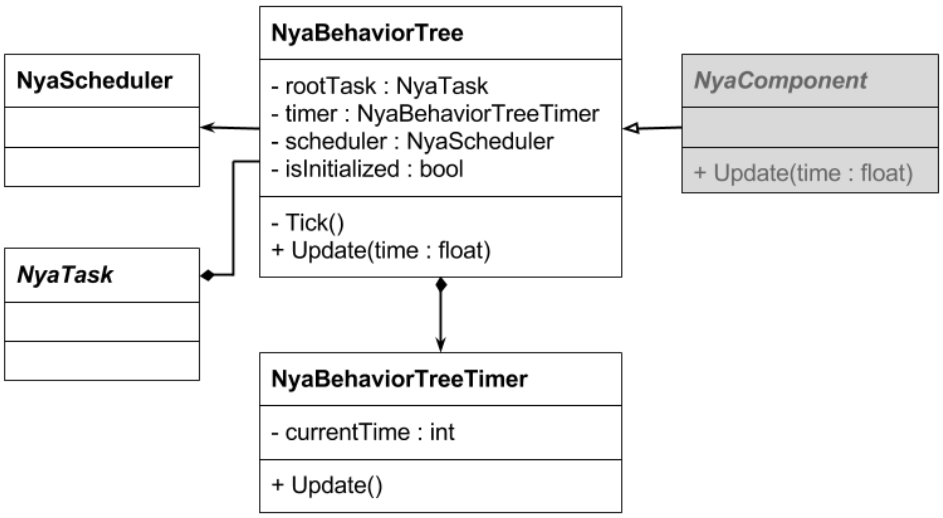
Kuvio 12. NyaScheduler-luokan luokkakaavio

Kun NyaScheduler-luokka käy runningTasks-listan läpi, kutsuu se kaikkien listassa olevien NyaTaskEntry-olioiden sisään tallennettujen NyaTask-olioiden Execute()-metodeita. Mikäli metodi antaa palautusarvoksi jotain muuta kuin NyaTaskReturnState.Running, todetaan tehtävän suoritus päättyneeksi, ja NyaScheduler-luokka ilmoittaa tehtävän päättymisestä tehtävästä kiinnostuneille tarkkailijoille - käytännössä NyaSchedulerObserver- ja NyaTaskObserver-olioille - kutsumalla niiden OnTaskFinished()-metodia. Lopulta completedTasks-listaa apuna käyttäen valmistuneen tehtävän NyaSchedulerEntry-olio hävitetään. (Kuvio 12.)

NyaScheduler on tehty liitettäväksi suoraan NyaEngine-pelimoottoriin. Se on peritty NyaModule-luokasta, ja voidaan siten voidaan lisätä pelimoottorin eri järjestelmiä automaattisesti suorittavaan NyaGameInstance-olioon. NyaGameInstance-olio päivittää NyaScheduler-oliota kutsumalla sen Update()-metodia. (Kuvio 12.)

NyaBehaviorTree

NyaBehaviorTree on luokka, joka kuvastaa varsinaista pelihahmolla olevaa käytöspuuta. Se sisältää rootTask-muuttujaan tallennettuna tehtävähirarkian, joka koostuu NyaTask-luokasta perityistä tehtävistä. Puu rakennettaan siis luokan ulkopuolella, ja rakennetun puun juuritehtävä asetetaan rootTask-muuttujaan. Käytöspuuta päivitetään askeleittain kutsumalla Tick()-metodia. Metodi selvittää mitkä puussa olevat tehtävät haluavat aktivoitua kutsumalla juuritehtävän QueryActivation()-metodia, ja jättää sitten varsinaisen aktiivisen reitin valinnan itse tehtäväluokille. (Kuvio 13.)



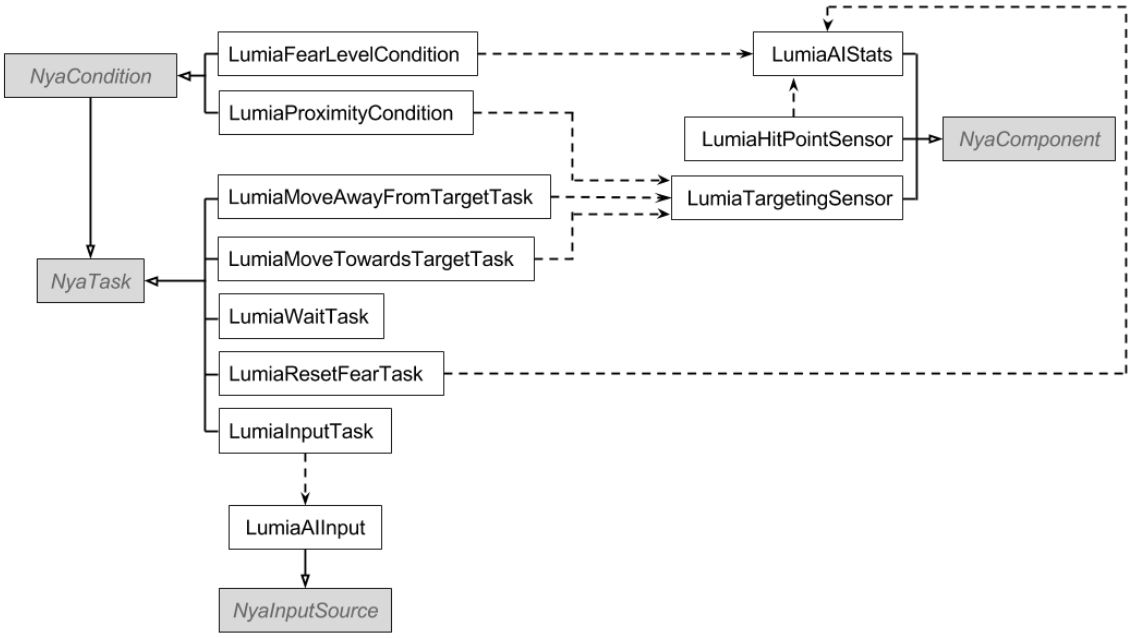
Kuvio 13. NyaBehaviorTree-luokan luokkakaavio

NyaBehaviorTree sitoo myös käytöspuujärjestelmän NyaEngine-pelimoottoriin, sillä se on peritty moottoriin kuuluvasta NyaComponent-luokasta. NyaComponent-luokan olio voidaan lisätä mihin tahansa moottorin NyaGameObject-olioon, eli käytännössä mihin tahansa pelihahmoon. Koska NyaBehaviorTree-luokan Update()-metodi on toteutettu uudelleen NyaComponent()-luokan vastaavasta virtuaalisesta metodista, päivittää moottori NyaBehaviorTree-luokkaa automaattisesti. (Kuvio 13.)

NyaBehaviorTree käyttää kuitenkin omaa moottorista erillistä ajoitusjärjestelmäänsä, joka sekunteissa mitattavan ajan sijaan käyttää pelin päivityskehyksiä (update frame). Ajoitusjärjestelmä on toteutettu äärimmäisen yksinkertaiseen NyaBehaviorTreeTimer-ajastinluokkaan, jota päivitetään myös Update()-metodissa. Ajastin sisältää käytännössä vain yhden juoksevan kokonaisluvun, ja ajastimeen päästään käsiksi mistä tahansa tehtävästä NyaBehaviorTree-luokan kautta.

6.2.2 Access Lumia –pelin tekoälyosat (LumiaEngine.AI-nimiavaruus)

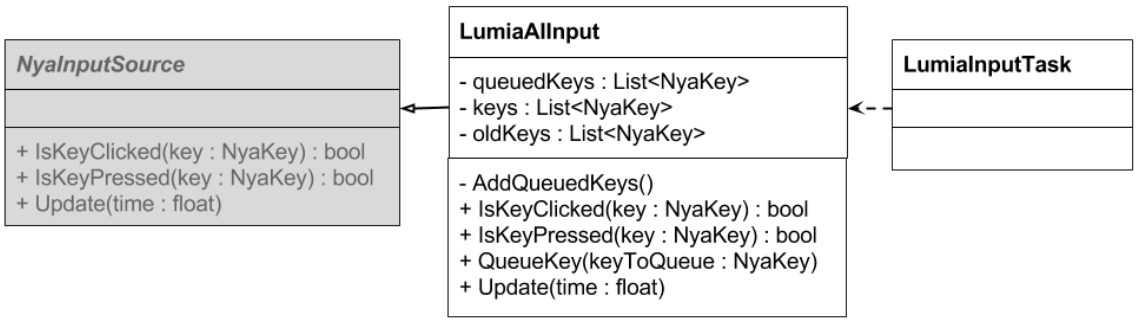
LumiaEngine.AI-nimiavaruuteen toteutetut luokat toteuttavat pelin tekoälyn vaatimat yksilölliset toiminnallisuudet. Luokat käyttävät laajasti pohjanaan NyaBehaviorTreeEngine-järjestelmän abstrakteja luokkia (Kuvio 14). Nimiavaruuden luokilla tekoälyhahmo saadaan liikkumaan, hyökkäämään ja ylipäätään käyttäytymään vaatimusmäärittelyssä esitetyllä tavalla.



Kuvio 14. Access Lumia –pelin tekoälyosien arkkitehtuuri

LumiaAIInput

LumiaAIInput on syötettä (input) käsittelevä luokka, johon tekoälyjärjestelmä - erityisesti LumiaInputTask-tehtäväluokka - voi syöttää syötekomentoja, kuten näppäinten painalluksia. Koska luokka on peritty NyaEngine-pelimoottoriin kuuluvasta NyaInputSource-luokasta, voidaan siihen talletetut syötteet lukea tarvittavissa paikoissa eri järjestelmissä, ja hoituu luokan päivitys automaattisesti pelimoottorin syötejärjestelmässä. LumiaAIInput-luokka huolehtii siitä, että tekoäly on pelaajan kanssa tasavertaisessa asemassa näppäinkomentoja syötettäessä, eikä tekoäly voi (tahallaan tai tahattomasti) huijata tekemällä toimintoja, jotka olisivat pelaajalle mahdottomia.

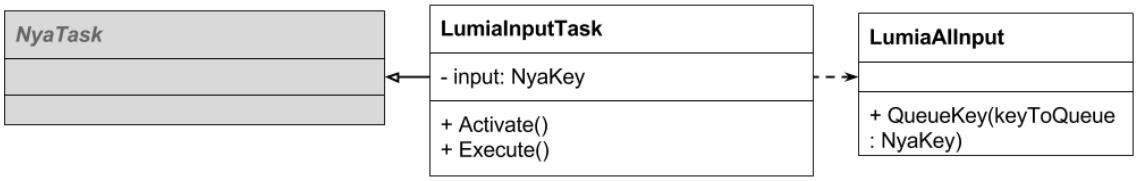


Kuvio 15. LumiaAIInput-luokan luokkakaavio

Luokkaan voidaan lisätä suoritettuja näppäinpainalluksia lisäämällä siihen painettua näppäintä vastaava NyaKey-arvo luokan QueueKey()-metodilla. Syötetty arvo lisätään keys-listaan seuraavalla Update()-metodin kutsulla. Painallus on listassa voimassa yhden päivityskehyksen ajan, ja näppäinten tilaa voidaan kysyä NyaInputSource-luokan virtuaalisista metodeista uudelleen toteutetuilla IsKeyClicked()- ja IsKeyPressed()-metodeilla. IsKeyClicked()-metodilla tarkistetaan, onko haluttu näppäin painettu juuri pohjaan, kun IsKeyPressed()-metodi vuorostaan kertoo, onko näppäin ollut kenties pohjassa jo jonkin aikaa. Metodit hyödyntävät keys- ja oldKeys-listojen tietoa nykyisestä ja yhtä päivityskehystä aiemmasta näppäimistön tilasta palautusarvojensa päättämisessä.

LumiaInputTask

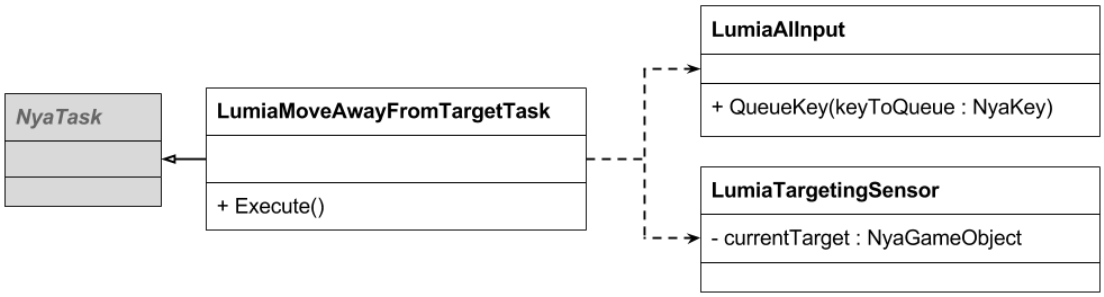
LumiaInputTask on NyaTask-luokasta peritty tehtäväluokka, joka lähettää yksittäisen näppäinkomennon tekoälyn ohjaamalle hahmolle. Se syöttää input-muuttujaansa tallennettun näppäinkomennon tekoälyn syötettä vastaanottavaan LumiaAIInput-olioon. Luokka vaatii, että tekoälyn ohjaamaan hahmoon on liitetty Access Lumia -pelin hahmojen syötettä tulkkaava osa, LumiaInputController, jonka syötelähteenä toimivaan inputSource-muuttujaan on asetettu LumiaAIInput-tyypin olio. NyaTask-luokan virtuaalisesta metodista uudelleenkirjoitetussa Execute()-metodissa LumiaInputTask-luokka yksinkertaisesti syöttää näppäinkomennon LumiaAIInput-luokan QueueKey()-metodin kautta. Tehtävä valmistuu heti, kun näppäinkomento on syötetty ja palauttaa arvon NyaReturnState.Completed. (Kuvio 16.)



Kuvio 16. LumiaInputTask-luokan luokkakaavio

LumiaMoveAwayFromTargetTask

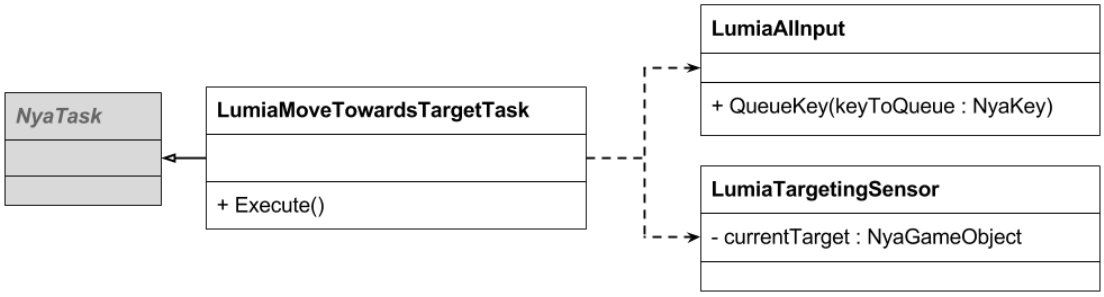
LumiaMoveAwayFromTargetTask on NyaTask-luokasta peritty tehtäväluokka, joka saa tekoälyhahmon juoksemaan poispäin tekoälyn kohteena olevasta hahmosta. Kohde saadaan LumiaTargetingSensor-luokasta, ja siten luokka vaatiikin, että tekoälyhahmolle on liitetty LumiaTargetingSensor-olio. Itse liikkuminen saadaan aikaan syöttämällä oikea näppäinkomento (vasemmalle tai oikealle riippuen siitä, kummalla puolella kohde on) hahmon LumiaAIInput-olion QueueKey()-metodin kautta. Siten LumiaMoveAwayFromTargetTask-luokka, kuten LumiaInputTask-luokkakin, vaatii myös, että hahmoon on liitetty LumiaInputController-luokka, jonka syötelähteenä on LumiaAIInput-olio. Tehtävä ei valmistu ikinä, ja luottaakin siihen että jokin sen ulkopuolinen tehtävä keskeyttää sen suorituksen. (Kuvio 17.)



Kuvio 17. LumiaMoveAwayFromTargetTask-luokan luokkakaavio.

LumiaMoveTowardsTargetTask

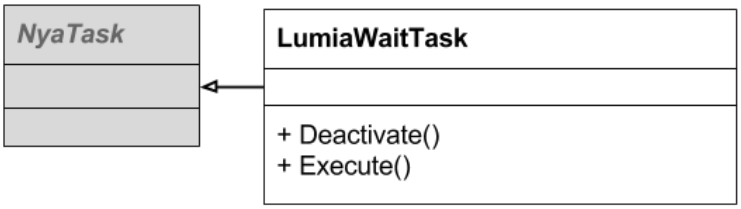
LumiaMoveTowardsTargetTask on NyaTask-luokasta peritty tehtäväluokka, joka saa tekoälyhahmon juoksemaan sen kohteena olevaa hahmoa kohti. Käytännössä luokka on identtinen LumiaMoveAwayFromTargetTask-luokan kanssa, mutta liikkumisen näppäinpainallukset on vaihdettu keskenään (Kuvio 18). Luokilla on myös samat vaatimukset tekoälyhahmoon liitettyjen osien kannalta. Nämä kaksi luokkaa olisi voitu pitää yhtenä luokkana, jossa liikkeen suunta olisi tallennettu muuttujaan, mutta päädyttiin erottelemaan omiksi luokikseen, että tekoälylle annettavat komennot pysyisivät mahdollisimman selkeinä ja yksinkertaisina. Lisäksi erottelu mahdollistaa sen, että järjestelmää laajennettaessa opinnäytetyön ulkopuolella voidaan kummallekin liikkumistyypille määritellä omaa, toisistaan eriävää logiikkaa.



Kuvio 18. LumiaMoveTowardsTargetTask-luokan luokkakaavio

LumiaWaitTask

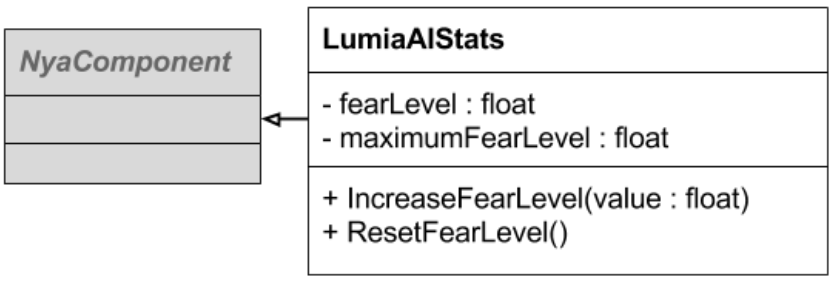
LumiaWaitTask on NyaTask-luokasta peritty tehtäväluokka, jonka tarkoituksena on saada tekoälyhahmo odottamaan paikoillaan määritellyn ajan. Aika ei ole sekunteina, vaan päivitys-kehyksinä, ja se asetetaan luokan framesToWait-muuttujaan. Jokaisella Execute()-metodin kutsulla frameTimer-muuttujaa lisätään yhdellä, ja kun se saavuttaa framesToWait-muuttujan arvon, tehtävä valmistuu. (Kuvio 19.)



Kuvio 19. LumiaWaitTask-luokan luokkakaavio

LumiaAIStats

LumiaAIStats on NyaComponent-luokasta peritty luokka, joka liitetään tekoälyhahmoon ja sisältää tekoälyn toimintaan käytettäviä arvoja. Luokkaa voidaan pitää eräänlaisena säiliönä muuttujille, joiden avulla tekoälyn toimintaa voidaan muokata. Opinnäytetyön laajuudessa luokalla on vain yksi oleellinen arvo: pelkotaso. Pelkotasoa, joka on tallennettu luokan fear-Level-muuttujaan, käytetään päättämään, milloin tekoälyhahmon tulee lähteä pakoon. Pelko-tasoa voidaan lisätä kutsumalla luokan IncreaseFearLevel()-metodia, ja se voidaan nollata ResetFearLevel()-metodilla. (Kuvio 20.)

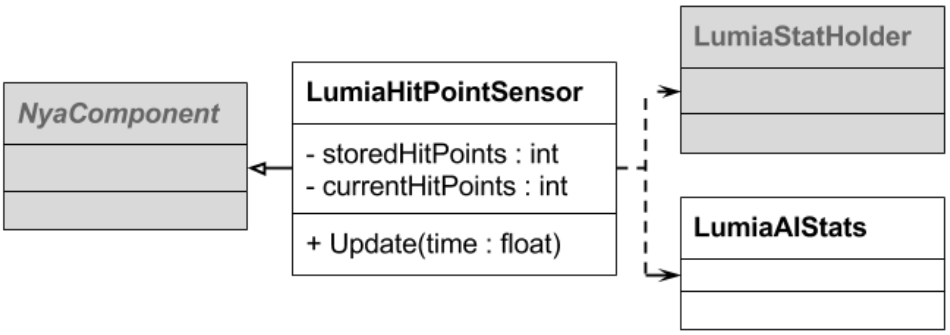


Kuvio 20. LumiaAIStats-luokan luokkakaavio

LumiaAIStats-luokkaa voitaisiin laajentaa helposti lisäämällä siihen uusia muuttujia. Mikäli muuttujia tarvitaan huomattavasti suurempia määriä, voitaisiin muuttujien hallintaan kehittää luokan sisälle oma järjestelmänsä, joka mahdollistaisi rajattoman määrän muuttujia ilman, että luokkaan tarvitsisi lisätä koodia. Tämän opinnäytetyön rajoissa yksinkertainen toteutus oli kuitenkin riittävä.

LumiaHitPointSensor

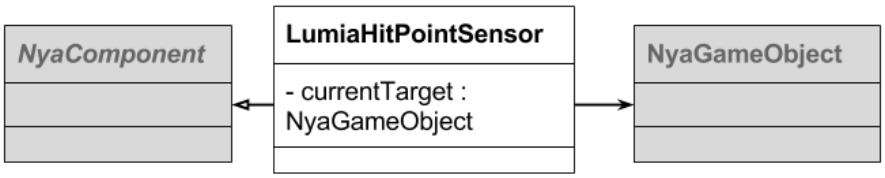
LumiaHitPointSensor on NyaComponent-luokasta peritty luokka, joka tekoälyhahmoon liitettynä seuraa hahmon elämäpisteitä (hit points) ja lisää hahmon pelkotasoa, kun elämäpisteet vähenevät. Luokka vaatii, että hahmolle on liitetty Access Lumia -pelin elämäpisteet sisältävä LumiaStatHolder-olio, sekä pelkotason sisältävä LumiaAIStats-olio. LumiaHitPointSensor mahdollistaa sen, että tekoälyhahmo lähtee pakoon, kun se on ottanut tarpeeksi vahinkoa. (Kuvio 21.)



Kuvio 21. LumiaHitPointSensor-luokan luokkakaavio

LumiaTargetingSensor

LumiaTargetingSensor on NyaComponent-luokasta peritty osa, joka pitää kirjaa siitä, mikä pelihahmo on milläkin hetkellä tekoälyhahmon kohteena. Kohde asetetaan luokan currentTarget-muuttujaan (Kuvio 22), ja muut järjestelmät voivat luokan kautta käyttää yhteistä kohdetta toimintoihinsa, kuten hyökkäämiseen tai kohdetta kohti liikkumiseen.

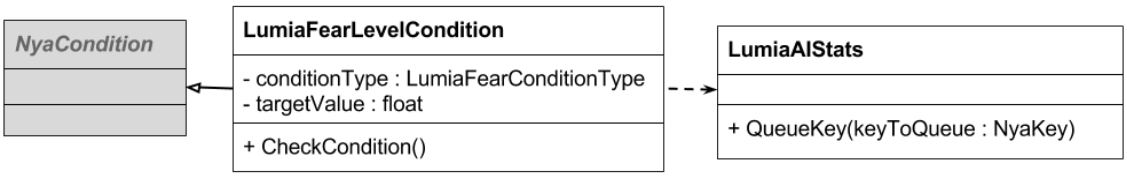


Kuvio 22. LumiaTargetingSensor-luokan luokkakaavio

LumiaTargetingSensor-luokkaa voitaisiin laajentaa hakemaan esimerkiksi lähin kohde useasta vaihtoehtoisesta kohteesta reaaliaikaisesti. Tämän opinnäytetyön kannalta kohteen käsin syöttäminen oli kuitenkin riittävää, sillä opinnäytetyövaiheessa pelissä oli vain yksi pelaaja ja yksi tekoälyhahmon ohjaama hahmo.

LumiaFearLevelCondition

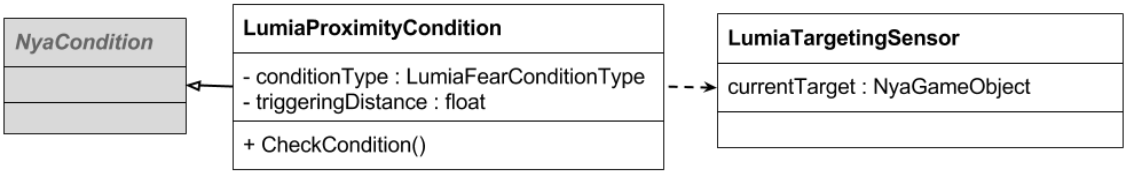
LumiaFearLevelCondition on tehtäväluokka, joka on peritty ehtoja kuvaavasta NyaCondition-luokasta. Luokka tarkistaa, onko tekoälyhahmon pelkotaso jonkin asetetun tason yläpuolella, alapuolella tai täsmälleen tietyssä arvossa sen mukaan, mikä arvotyyppi on asetettu luokan conditionType-muuttujaan. Luokalle asetetaan vertailuun haluttu arvo sen targetValue-muuttujaan. Koska LumiaFearLevelCondition käsittelee pelkotasoa, täytyy sen tarkkailemalle tekoälyhahmolle olla liitetty LumiaAIStats-olio. (Kuvio 23.)



Kuvio 23. LumiaFearLevelCondition-luokan luokkakaavio

LumiaProximityCondition

LumiaProximityCondition on NyaCondition-luokasta peritty ehto, joka tarkistaa onko tekoälyhahmon kohteena oleva hahmo lähempänä tai kauempana kuin ehdon triggeringDistance-muuttujaan määritelty etäisyys. Tarkistamisen tyyppi riippuu siitä, kumpi tapa on asetettu luokan conditionType-muuttujaan. Vertailua varten LumiaProximityCondition vaatii, että tekoälyhahmoon on liitetty kohteen sisällään pitävä LumiaTargetingSensor-olio. (Kuvio 24.)

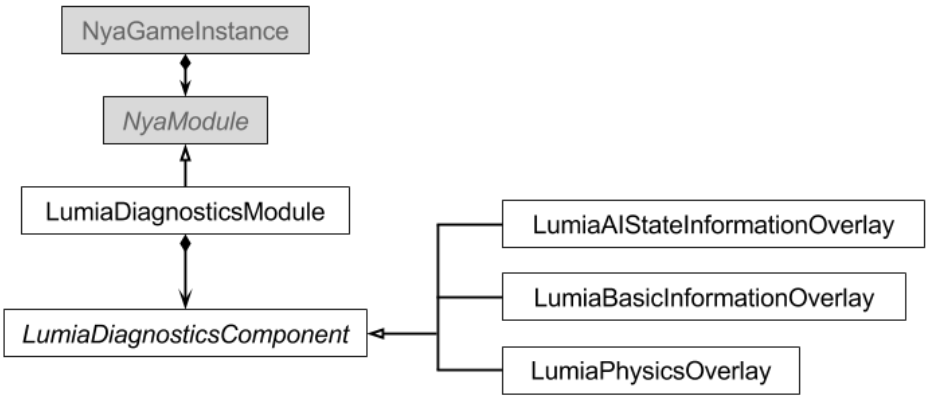


Kuvio 24. LumiaProximityCondition-luokan luokkakaavio

Luokkaa käytetään opinnäytetyön käytöspuussa kahdessa eri tilanteessa. Kun tekoälyhahmo on pakenemistilassa, tarkistetaan, onko hahmo päässyt tarpeeksi kauas pelaajahahmosta, että pakenemistila voidaan lopettaa. Kun tekoäly haluaa hyökätä, se tarkistaa onko pelaaja tarpeeksi lähellä, että hyökkäys voidaan aloittaa.

6.2.3 Seurantajärjestelmä (LumiaDiagnostics)

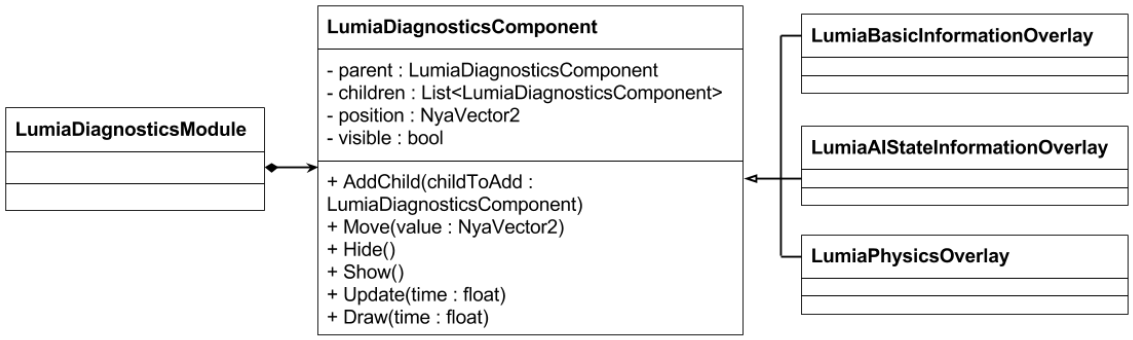
LumiaDiagnostics on Access Lumia -peliin kehitetty kirjasto, joka mahdollistaa tiedon näyttämisen ruudulla omissa yksilöllisissä, helposti vaihdettavissa näkymissään. Kirjaston osat on erityisesti toteutettu näyttämään tekoälyjärjestelmän kannalta oleellista tietoa, ja sitä käytetään etsimään ongelmia ja korjaamaan virheitä seuratuissa järjestelmissä. LumiaDiagnostics toimii lisäosana peliin ja sen alla toimivaan NyaEngine-pelimoottoriin. (Kuvio 25.)



Kuvio 25. LumiaDiagnostics-seurantajärjestelmän arkkitehtuuri

LumiaDiagnosticsComponent

LumiaDiagnosticsComponent on abstraktia seurantatiedon näyttämiseen tarkoitettua käyttöliittymän osaa kuvaava luokka. Luokan olioilla voi olla vanhempi tai lapsia, eli luokalla voidaan rakentaa käyttöliittymään hierarkioita. LumiaDiagnosticsComponent-luokasta voidaan periä mitä tahansa tarvittavia käyttöliittymäosia, kuten ikkunoita, jotka sisältävät lukuisia eri osia, tai esimerkiksi yksinkertaisia tekstikenttiä.

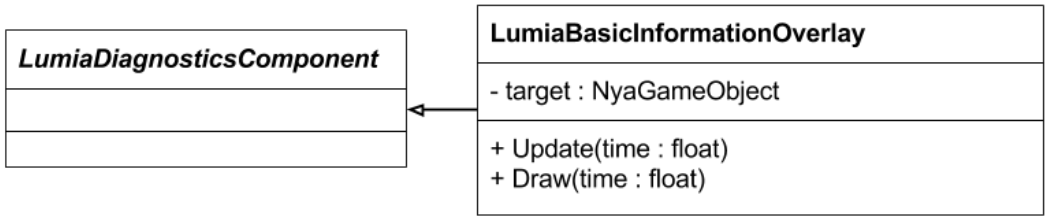


Kuvio 26. LumiaDiagnosticsComponent-luokan luokkakaavio

LumiaDiagnosticsComponent-luokka sisältää käyttöliittymät osan näyttämiseen ja piilottamiseen Hide()- ja Show()-metodien kautta, sekä osan siirtämiseen position-muuttujan ja Move()-metodin kautta. Näyttäminen, piilottaminen ja siirtäminen aiheuttavat saman toimenpiteen tekemisen myös kaikille olion lapsiosille. Luokka hoitaa myös lapsiensa piirtämisen ja päivittämisen. (Kuvio 26.)

LumiaBasicInformationOverlay

LumiaBasicInformationOverlay on LumiaDiagnosticsComponent-luokasta peritty näkymä, joka näyttää perustason tietoa sen kohteena target-muuttujassa olevasta hahmosta (Kuvio 27). Perustietoon kuuluvat hahmon nimi, reaaliaikainen sijainti ja nopeus sekä animaatiotila. Näkymä toimii viitekehyksenä muiden näkymien kanssa ongelmien etsimisessä luvun 5.4.2 esittämällä tavalla.

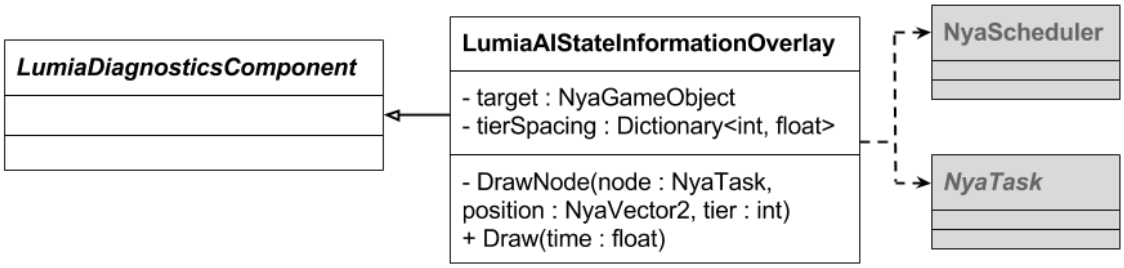


Kuvio 27. LumiaBasicInformationOverlay-luokan luokkakaavio

LumiaAIStateInformationOverlay

LumiaAIStateInformationOverlay on LumiaDiagnosticsComponent-luokasta peritty näkymä, joka näyttää tekoälyn käytöspuun tilan reaaliaikaisesti. Luokka näyttää kaikki puun jäsenet, eli käytännössä tehtävät, ja erottelee puun aktiivisina NyaScheduler-aikatauluttajaluokassa suoritettavina olevat osat. LumiaAIStateInformationOverlay-näkymän avulla voidaan helposti seurata, kuljetaanko käytöspuussa loogisella tavalla, vai tapahtuuko puussa jotain, mitä ei pitäisi. Näkymä on ehdottomasti tärkein seurantanäkymä tekoälyjärjestelmän kannalta.

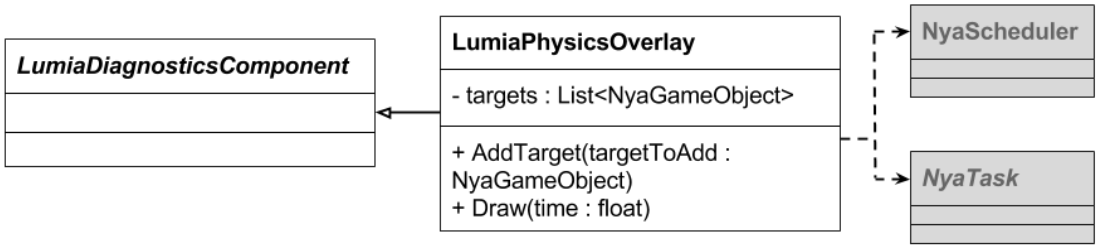
LumiaAIStateInformationOverlay-luokka piirtää Draw()-metodissaan koko sen target-muuttujaan liitetyn tekoälyhahmon käytöspuun juuritehtävästä alkaen. Tehtävät piirretään neliöinä, joiden sisällä on tehtävän nimi. Eri hierarkian tasot piirretään sopivasti aseteltuna ylhäältä alas. Luokassa on myös mahdollisuus lisätä eri kerroksissa olevien tehtävälaatikoiden piirtoetäisyyttä toisistaan SetTierSpacing()-metodin kautta. Metodia käyttämällä voidaan pääasiassa välttää päällekkäin piirtyviä tehtävälaatikoita. Puun aktiiviset tehtävät saadaan tietoon NyaScheduler-luokan staattisen GetGameObjectTasks()-metodin kautta, joka palauttaa kaikki aikatauluttajassa olevat halutun pelihahmon tehtävät. Aktiiviseksi todetut tehtävät piirretään sitten eri värillä, että ne voidaan helposti erottaa näkymässä. (Kuvio 28.)



Kuvio 28. LumiaAIStateInformationOverlay-luokan luokkakaavio

LumiaPhysicsOverlay

LumiaPhysicsOverlay on LumiaDiagnosticsComponent-luokasta peritty näkymä, joka näyttää reaaliaikaisesti sen targets-listaan asetettujen hahmojen törmäystarkastusympyrät läpinäkyvinä, värjättyinä ympyröinä hahmojen päällä. Hyökkäävät ympyrät, jotka aiheuttavat vahinkoa osuessaan puolustaviin ympyröihin, on värjätty punaisiksi, ja vahinkoa vastaanottavat puolustavat ympyrät on värjätty sinisiksi. Luokkaan voidaan lisätä niin monta hahmoa kuin halutaan, ja se piirtää kaikkien lisättyjen hahmojen törmäystarkastusympyrät. LumiaPhysicsOverlay vaatii, että hahmolle on lisätty Access Lumia-pelin taisteluanimaatioita hallitseva LumiaAnimationController-olio. (Kuvio 29.)

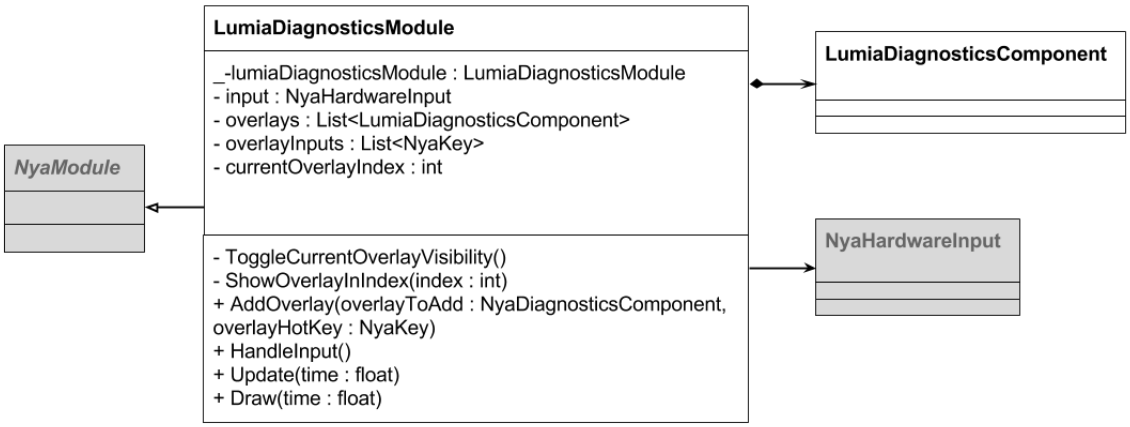


Kuvio 29. LumiaPhysicsOverlay-luokan luokkakaavio

LumiaPhysicsOverlay-luokalla voidaan helposti havaita virheitä törmäystarkastusympyröiden asetteluissa tai ominaisuuksissa. Näkymän avulla on helppo havaita, mikäli jostain animaatiokehyksestä puuttuu jokin ympyrä, tai jos ympyrä on vahingossa vääräntyyppinen. Vaikka näkymä liittyy enemmän fysiikkaan kuin tekoälyyn, on se taistelupelin kaltaisessa pelissä hyvin tiiviisti sidoksissa tekoälyyn, koska tekoälyn tulee pystyä hyökkäämään onnistuneesti kohdettaan vastaan.

LumiaDiagnosticsModule

LumiaDiagnosticsModule on NyaModule-luokasta peritty osa, joka hallitsee ja piirtää seurantajärjestelmän näkymiä ja mahdollistaa näkymästä toiseen vaihtamisen näppäinpainalluksilla. Koska luokka on peritty NyaModule-luokasta, se myös sitoo seurantajärjestelmän NyaEngine-pelimoottoriin, joka pystyy hoitamaan luokan piirron ja päivittämisen automaattisesti.

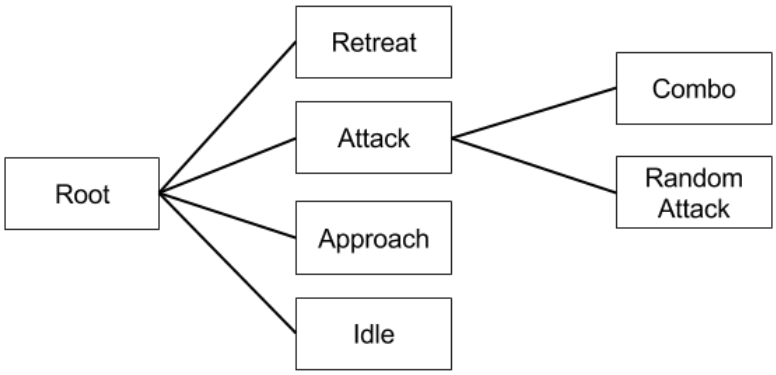


Kuvio 30. LumiaDiagnosticsModule-luokan luokkakaavio

LumiaDiagnosticsModule-luokkaan voidaan lisätä LumiaDiagnosticsComponent-luokasta peritty näkymä AddOverlay()-metodin kautta. Samalla syötetään näppäinkomento, jolla kyseinen näkymä saadaan näkyviin - tai pois näkyvistä, jos näkymä on jo aktiivisena. Samaa näppäinkomentoa ei saa lisätä luokkaan kahta kertaa. Luokka käyttää LumiaDiagnosticsComponent-luokan Show()- ja Hide()-metodeita eri näkymien näyttämiseen ja piilottamiseen. (Kuvio 30.)

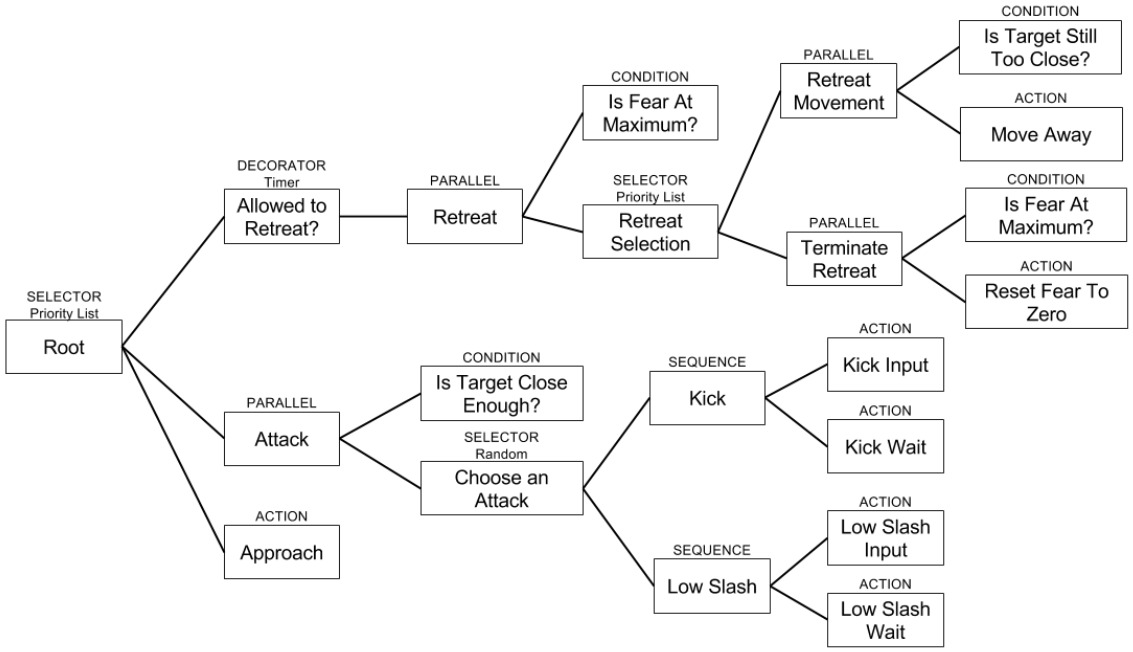
6.3 Toteutus

Käytöspuuta luodessa puuhun piti lähtökohtaisesti saada näkymään tekoälyn käytöksen eri tilat: liikkuminen, hyökkääminen, pakeneminen ja odottelu. Voitiin huomata, että tiloilla oli selkeä tärkeysjärjestys. Pakeneminen oli kaikkein tärkein, koska kun pakenemista rajoittavat ehdot täyttyivät, ei tekoäly järkevästi voinut jäädä hyökkäämään tai odottelemaan. Seuraavana oli hyökkääminen, koska myös hyökkäämisellä oli omat rajoittavat ehtonsa: kohteen piti olla tarpeeksi lähellä. Oli myös selvää, että jos voitiin valita hyökkäämisen ja liikkumisen välillä, ja koska pelin tarkoituksena oli saada toinen pelaaja pois pelistä hyökkäämällä, oli hyödyllisempää, että tekoäly yrittäisi voittaa pelaajan aina kun mahdollista. Samoin liikkuminen oli selvästi odottelua järkevämpi vaihtoehto, sillä odottelua tehtäisiin vain kun mitään muuta ei ole tehtävissä. (Kuvio 31.)



Kuvio 31. Access Lumia –pelin käytöspuun eri tilat

Tilojen luomisen jälkeen käytöspuusta puuttuivat kuitenkin säännöt, millä tilojen tulisi vaihtua. Esimerkiksi pakenemistilaan tulisi joutua vain, kun tekoälyhahmon pelkotaso on noussut tarpeeksi korkealle. Pakeneminen on myös sallittua vain kerran tietyssä ajassa. Jouduttiin siis miettimään, millä rakenteilla nämä säännöt saataisiin puuhun rakennettua. Voitiin kuitenkin nopeasti huomata, että kaikille tapauksille oli jo omat, luonnollisesti tarkoituksiin sopivat rakenteensa luvun 5 mukaan toteutetuissa käytöspuun osissa. NyaCondition-tyyppiset ehdot toimivat erinomaisesti esimerkiksi NyaSelector-valitsijan tai NyaSequence-tehtäväsarjan kanssa, kun osat yhdistetään NyaParallel-tyyppisen rinnakkaisesti suoritettavan tehtäväsarjan avulla. Käytännössä haluttua toimintoa tehtäisiin siis vain, jos kyseinen ehto täyttyisi. Pakenemisen rajoittaminen vain tietyin aikavälein tapahtuvaksi onnistui hyvin NyaTimerDecorator-tyyppisellä koristelijalla.



Kuvio 32. Access Lumia –pelin lopullinen käytöspuu

Lopulta saatiin aikaan täysin valmis käytöspuu, jonka kaikille haaroille on määritelty niiden tehtävätyyppi (Kuvio 32). Tällaisessa muodossa käytöspuu pystyttiin jo toteuttamaan kooditasolla. Lopullinen käytöspuu ja luvussa 5 kuvaillut osat muodostuivat kuitenkin yhtä aikaa, eikä niitä tehty toisistaan erillisinä kokonaisuuksina, koska tarvittavien rakenteiden hahmottaminen kummassakin tapauksessa olisi ollut liian hankalaa. LumiaEngine.AI-nimiavaruuden osat toteutettiin vasta käytöspuun hahmottelun jälkeen, koska ne käytännössä vain toteuttavat käytöspuun vaatimat osat.

Lopullista käytöspuuta tehdessä havaittiin myös ongelmia, jotka muokkasivat puuta merkittävästikin. Vaatimusmäärittelyssäkin keskeisinä olleista liikesarjoista jouduttiin luopumaan jo toteutusvaiheessa, koska paljastui, että alla olevista Access Lumia -pelin animaatio- ja hyökkäysjärjestelmistä oli käytännössä mahdotonta saada tietoa siitä, milloin hahmon tekemä hyökkäys alkaa tai päättyy. Ominaisuuden lisääminen olisi ollut opinnäytetyön mittakaavassa liian työlästä, joten liikesarjat päätettiin jättää pois erityisenä ominaisuutena.

Seurantajärjestelmä toteutettiin, kun käytöspuun viimeiset osat saatiin valmiiksi. Järjestelmän ohjelmointi oli suoraviivaista, ja enemmän huomioita nousi seurantajärjestelmää ohjelmoitaessa itse tekoälyjärjestelmässä - osoittaen jo heti alkuvaiheessa seurantajärjestelmän hyödyllisyyden. Käytöspuun tilan reaaliaikaisesti piirtävästä LumiaAIStateInformationOverlay-näkymästä havaittiin välittömästi, että käytöspuun osia jäi NyaScheduler-aikatauluttajaan

päälle myös silloin, kun käytöspuun aktiivinen reitti oli siirtynyt toisaalle. Vika pystyttiin jäljittämään käytöspuun perusosiin, erityisesti NyaSequence- ja NyaSelector-luokkiin, ja korjattua viipymättä.

Kehityksen loppuvaiheilla myös huomattiin, että usein liikkuessaan hyökkäämään tekoälyhahmo ei liikkunutkaan tarpeeksi lähelle, että hyökkäykset osuisivat. Sama ongelma jatkui siitäkin huolimatta, että hyökkäysetäisyyttä säädettiin yhä pienemmäksi ja pienemmäksi. LumiaPhysicsInformationOverlay-näkymästä voitiinkin havaita selkeästi, että hahmojen törmäystarkastusmuodot kyllä osuivat toisiinsa, ja ongelmana ei ollutkaan se, että tekoälyhahmo ei liikkunut tarpeeksi lähelle. Lopulta paljastui, että ongelma oli Access Lumia -pelin fysiikkajärjestelmässä. Tapaus toimii hyvänä esimerkkinä siitä, miten muut järjestelmät vaikuttavat olennaisesti tekoälyyn, ja siitä miten muiden järjestelmien toiminta vaikuttaa tekoälyn toiminnan havainnointiin. Tapaus myös osoitti vahvasti sen, että seurantajärjestelmässä kannattaa olla myös muita kuin tekoälyyn suoraan liittyviä osia tutkivia näkymiä.

6.4 Testaus

Tekoälyn testaus on toteutettu pääosin pelaamalla peliä ja tarkkailemalla tekoälyn toimintaa. Koska kyseessä on hyvin yksinkertainen tekoäly, ja tarkoituksena ei ole tehdä minkäänlaista tutkimusta, tekoälyn toimintaa ei testata ulkopuolisilla käyttäjillä. Se, kuinka hyvin tekoäly täyttää määritellyt vaatimukset, pohjataan täysin omiin havaintoihin. Tekoälyn toiminnassa havaituista ongelmista mainitaan kuinka ongelmat saadaan peliä pelaamalla toistettua.

Tekoäly osaa liikkua määritysten mukaisesti pelaajaa kohti ja pelaajasta poispäin. Tekoäly osaa myös havaita, onko se tarpeeksi lähellä tai kaukana pelaajasta. Tekoäly ei kuitenkaan osaa käyttää ilmatilaa hyväkseen, eikä osaa siten hypätä. Hyppytilanteessa hypyn kattama etäisyys ja laskeutumispaikka oli toteutuksessa liian hankalaa hahmottaa tai laskea, ja sitten tekoälyhahmon toiminnan sujuvuuden varmistamiseksi hyppiminen jätettiin pois.

Tekoäly pystyy hyökkäämään pelaajaan satunnaisesti valituilla hyökkäyksillä. Hyökkääminen on toteutettu määritelmien mukaisesti lähettämällä ulos näppäinsyötteitä, joilla hyökkäykset saadaan aikaan. Monet vaatimusmäärittelyssä olleista hyökkäysominaisuuksista eivät kuitenkaan siitä huolimatta täyttyneet aikarajoitusten ja teknisten ongelmien takia. Tekoäly ei osaa käyttää liikesarjoja, koska alla olevasta Access Lumia -pelin hyökkäysjärjestelmästä ei saatu

helposti tietoa eri hyökkäysten alkamisesta ja loppumisesta. Tekoäly ei myöskään osaa tehdä ilmahyökkäyksiä, koska kuten edellä mainittiin, ilmassa tapahtuvan liikkeen liikerataa oli hankala hahmottaa.

Hyökkäämiseen jäi myös perustavammanlaatuisia ongelmia. Tekoäly ei aina hyökkää heti sen jälkeen, kun se on liikkunut pelaajaa kohti. Hahmojen liikkeen päättyessä hahmo jarruttaa hetken aikaa, ennen kuin se siirtyy tyypilliseen odotustilaan. Tekoäly ei osaa tunnistaa omaa animaatiotilaansa, ja yrittää siten tehdä hyökkäystään jo jarrutustilassa, eikä pelin säännöt salli tätä. Tekoälystä voidaan tällöin havaita, että se jää jarrutuksen jälkeen seisomaan vain hetkeksi paikoilleen, ja tämä odotus on itse asiassa hyökkäyksen jälkeen tapahtuva hyökkäyksen loppumista odottava tila. Ongelma voitaisiin ratkaista esimerkiksi lisäämällä käytöspuun hyökkäyksiin osa, joka varmistaisi, että hyökkäys voidaan käynnistää vasta, kun tekoäly on saavuttanut halutun tilan. Tämä voisi olla tekoälyn loogisuuden kannalta järkevää, mutta aiheuttaisi sen, että tekoäly ei hyökkää kovin sulavasti liikkeen jälkeen, koska sen täytyisi aina odottaa jarrutusanimaatio loppuun. Sulavuuden kannalta parempana vaihtoehtona olisi sallia hyökkääminen suoraan juoksu- tai jarrutustiloista.

Toinen tekoälyn hyökkäämisessä oleva ongelma ilmenee, mikäli pelaaja on tarpeeksi lähellä tekoälyhahmoa, että tekoäly voi hyökätä, mutta onkin tekoälyhahmon selkäpuolella. Tekoäly ei tarkista millään tavalla, onko pelaaja tekoälyhahmon edessä vai takana, ja jatkaa siten vain hyökkäämistä täysin väärään suuntaan. Virhe on yksinkertainen ajatteluvirhe tekoälyä toteutettaessa, ja on helppo korjata lisäämällä hyökkäysvaiheeseen yksinkertainen osa, joka kääntää tekoälyn ympäri, mikäli se havaitsee pelaajan olevan tekoälyhahmon selkäpuolella.

Tekoäly osaa perääntyä, kun hahmo ottaa tarpeeksi vahinkoa, ja sen pelkotaso vahingonoton seurauksena saavuttaa maksimin. Tekoäly osaa myös tunnistaa milloin se on saanut tarpeeksi etäisyyttä, ja jatkaa pelaajan kimppuun hyökkäämistä nollaamalla pelkotasonsa. Saavutettu perääntymiskäyttäytyminen on kyllä pääosin toimiva, mutta käyttäytymiseltään liian konemainen. Tekoäly lähtee aina karkuun samalla tavalla, eikä osaa esimerkiksi aloittaa pakenemista hypystä, tai yrittää keskeyttää pelaajan mahdollista hyökkäystä nopealla potkulla ennen pakenemisen aloittamista. Pakenemiskäyttäytyminen on siis suhteellisen ennustettavaa, ja periaatteessa pelaaja voi käyttää tätä hyväkseen ja hyökätä tekoälyn selkään. Nykyisellään liikkuminen pelissä on kuitenkin riittävän nopeaa, että pelaaja tarvitsee nopeat refleksit hyväksikäyttääkseen tilannetta. Tällöin pakenevaan tekoälyhahmoon osumisesta tuleekin haastavaa ja palkitsevaa, ja ominaisuutta voidaan ajatella positiivisena.

Pakenemiskäyttäytymisen keskeisenä ongelmana on se, ettei siinä otettu huomioon kentän reunoja tai sitä, että pelaaja voi lähteä seuraamaan tekoälyhahmoa. Tekoälyhahmo yrittää aina itsepäisesti päästä tarpeeksi kauas pelaajahahmosta, ja jos kentän reuna tulee vastaan, hahmo jää jumiin juoksemaan seinää päin loputtomasti. Ongelma havaitaan äärimmäisen helposti, jos pelaaja saa tekoälyhahmon pakenemistilan päälle silloin, kun tekoälyhahmo on aivan kentän reunan vieressä. Pakenemistilaan pitäisi tehdä poikkeus, mikäli hahmo osuu kentän reunaan. Voisi myös olla järkevää, että tekoäly ei edes yritä paeta, mikäli kentän reuna on liian lähellä.

Tekoäly osaa rytmittää käyttäytymistään yksinkertaisilla periaatteilla. Tekoäly pitää taukoja hyökkäystensä välillä, ja pakenee välillä, antaen pelaajalle tarvittaessa tilaa rauhoittua. Pakenemistilan rytmityksessä ongelmana on kuitenkin se, että tekoäly ei ikinä pakenemisen jälkeen jää hetkeksi miettimään, antaen välillä pelaajallekin mahdollisuutta olla hyökkäävänä osapuolena, vaan tekoäly tulee aina suoraan takaisin pelaajan kimppuun.

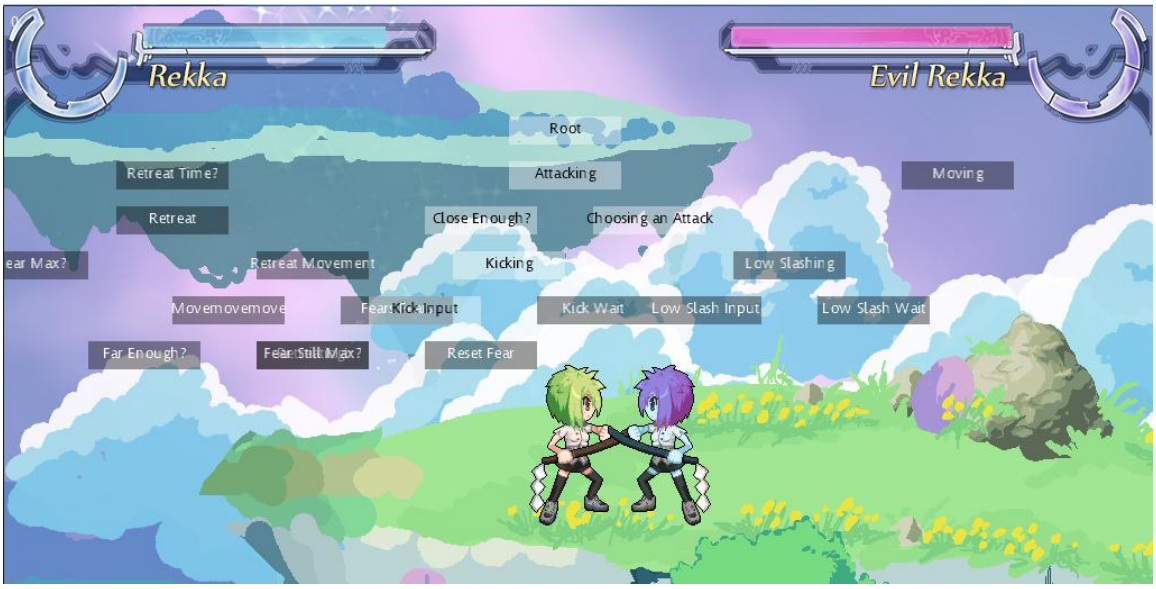
Tekoälyn vaikeustasosta ei saatu säädettävää. Tekoäly on toiminnaltaan aikarajoitteiden takia niin yksinkertainen, että siihen ei saatu järkeviä säädettäviä arvoja, joilla toiminta merkittävästi muuttuisi helpommaksi tai vaikeammaksi. Yksi harvoista säätömahdollisuuksista oli hyökkäysten välissä olevan odotusajan muuttaminen, mutta aikaa vähentämällä jouduttiin helposti tilaan, jossa tekoäly aloitti uuden hyökkäyksen tekemisen jo siinä vaiheessa, kun pelaaja oli vasta palautumassa edellisestä hyökkäyksestä. Tämä vuorostaan aiheutti sen, ettei pelaaja usein edes pystynyt reagoimaan hyökkäyksiin, ja pelaamisesta tuli näissä tilanteissa rasittavaa. Voitiin selkeästi huomata, että tekoälyn säädettäväksi tekeminen vaatisi huomattavasti jatkokehitystä.

Tekoälyjärjestelmän käytöspuuhun oli kehityksen aikana helppo lisätä uusia tiloja. Tilat pysyivät hyvin omina kokonaisuuksinaan, eivätkä ne ollut erityisesti sidoksissa toisiinsa, ellei sitä nimenomaan vaadittu. Tiloja voidaan lisätä täysin rajattomasti, ja toteutetut perusosat mahdollistavat toimivan käytöspuun tekemisen monenlaisiin pelityyppeihin ilman minkäänlaisia rajoituksia.

Seurantajärjestelmä täyttää erinomaisesti kaikki vaatimuksensa. Seurantajärjestelmän näkymät ovat täysin räätälöitävissä käyttäjän tarpeisiin, ja näkymien välillä voidaan helposti siirtyä näppäinkomennoilla. Uusia näkymiä voidaan järjestelmään lisätä täysin saumattomasti, ja vain eri näppäinkomentojen määrä rajoittaa näkymien lukumäärää. Seurantajärjestelmään

saatiin myös toteutettua kaikki Access Lumia -pelin tekoälyjärjestelmän seurantaan vaaditut näkymät. Näkymien toteutus vaati jonkin verran lisäyksiä itse pelin koodiin, että näkymiin tarvittavaan tietoon voitiin päästä käsiksi, mutta lisäykset olivat pieniä, eivätkä heikentäneet osien koodin laatua millään tavalla.

Ainoana ongelmana seurantajärjestelmässä oli näkymä, joka näytti pelin tekoälyhahmolla olevan käytöspuun. Käytöspuuhun tulee helposti niin paljon osia, ettei käytöspuuta saa hyvin mahtumaan pelin ikkunan sisään (Kuvio 33). Käytöspuun eri tehtävät menevät puussa myös helposti päällekkäin, jopa siitäkin huolimatta, että näkymään kehitettiin tapa, jolla samalla tasolla olevien tehtävien välimatkaa voitiin säätää. Näkymään voitaisiin jatkossa kehittää tapoja, joilla eri tehtävälaatikoita voitaisiin siirtää esimerkiksi hiirellä, ja käyttäjä voisi siten itse säätää puun näkymään sopivalla tavalla.



Kuvio 33. Tekoälyn käytöspuun piirtävä näkymä. Osa näkymän tehtävälaatikoista menee ruudun ulkopuolelle, ja osa menee keskenään päällekäin.

Kuten seurantajärjestelmän toteutuksen yhteydessä jo todettiin, seurantajärjestelmän eri näkymät täyttivät tehtävänsä hyvin jo heti niiden kehityksen alkuvaiheessa. Näkymillä löydettiin ongelmia niin tekoälyjärjestelmästä kuin muistakin tekoälyn toimintaan vaikuttavista Access Lumia -pelin järjestelmistä, ja virheet voitiin siten korjata välittömästi. Näkymiin käytetty aika ei myöskään ollut niin merkittävä, että se olisi haitannut millään tavalla tekoälyn parissa tehtyä työtä.

7 TULOKSET JA POHDINTA

Hyvien periaatteiden päälle rakennetut uudelleenkäytettävät perusosat mahdollistavat helposti laajennettavan tekoälyjärjestelmän luomisen. Juuri tämä tekee toteutetusta käytöspuujärjestelmästä erinomaisen toimivan. Yksilöllistä tekoälylogiikkaa sisältävä taistelupeli voidaan selvästikin toteuttaa käytöspuilla. Itse asiassa, sopivasti rakennettu käytöspuujärjestelmä ei ole millään tavalla riippuvainen tietystä pelityypistä, ja sopii rakenteensa vuoksi käytännössä minkälaiseen peliin tahansa. Huolenaihetta voi kuitenkin löytyä, jos eri käyttäytymisvaihtoehtoja vaaditaan suuria määriä, sillä silloin eri tehtäväluokkien määrä voi kasvaa niin suureksi, että niitä on hankala hallita. Aikarajoitusten takia tätä ei pystytty kuitenkin opinnäytetyössä varmistamaan, mutta asia on syytä ottaa huomioon, mikäli käytöspuujärjestelmällä lähdetään tekemään erittäin laajaa tekoälyä.

Tekoälyn tekemiseen liittyvä työmäärä olikin suurimpia yllätyksiä opinnäytetyötä tehdessä. Jo yksinkertainenkin tekoäly vaatii huomattavaa määrää työtä. Mikäli tekoälystä halutaan viihdyttävä vastustaja - joka peleissä on käytännössä ainoa tärkeä asia - pitää siihen varata paljon resursseja. Tuntuakseen älykkäältä, tekoäly vaatii paljon erilaista logiikkaa ja vaihtoehtoja käyttäytymiseensä, että se osaisi reagoida erilaisiin tilanteisiin. Erityisesti taistelupelissä tekoäly vaatii paljon erilaisia hyökkäyksiä, joilla on paljon vaihtelevia ominaisuuksia, jotka tekoälyn täytyy toiminnassaan ottaa huomioon. Koska erilaiset hyökkäykset vaativat myös samalla valtavan määrän grafiikkaa ja animaatioita, tulee kokonaistyömäärästä ylivoimainen haaste.

Työmäärään vaikuttaa myös se, että hyvä ja monipuolinen tekoäly vaatii paljon myös sen alla olevilta järjestelmiltä. Tekoälyn pitää päästä käsiksi monipuolisesti pelin eri järjestelmiin, kuten animaatiotiloihin, eri hyökkäyksien tietoihin ja pelikentän tietoihin, että se pystyy tekemään älykkäitä päätöksiä. Samoin alla olevia järjestelmiä ja pelin sisältöä voidaan joutua muokkaamaan, mikäli niiden vaikutus saa tekoälyn näyttämään peliä pelaavan havaitsijan näkökulmasta huonolta tai epäuskottavalta. Voi jopa olla, että tekoäly vaikuttaa suoraan pelin suunnitteluunkin. Esimerkiksi jos pelissä on valtavasti erilaisia, yksittäisiä hyökkäyksiä, voi käydä mahdottomaksi saada tekoäly reagoimaan jokaiseen näistä hyökkäyksistä oikein. Tällöin on tekoälyn kannalta välttämätöntä, että hyökkäykset yleistetään jonkinlaisiin ryhmiin, joihin tekoäly osaa reagoida ryhmälle tyypillisellä logiikalla.

Opinnäytetyössä tekoälyjärjestelmästä ei saatu riittävän datapainotteista. Yksinkertaisen tekoälyn käyttäytymiselle oli vaikea saada minkäänlaista järkevää säädettävyyttä. Tekoälyn vaikeustaso jäi melko matalalle, ja yritykset nostaa sitä johtivat vain pelaajan kannalta rasittavaan pelattavuuteen. Tekoälyä pitäisi monimutkaistaa huomattavasti, että sille voitaisiin saada kaupallisista taistelupeleistä tuttuja vaikeustaso-ominaisuuksia. Säädeltävyyttä varten tekoäly tarvitsisi lukuisia eri muuttujia, joista riippuvainen sen käytöksen tulisi olla. Pienessä projektissa tällaisen tekoälyn tekeminen on kuitenkin hyvin hankalaa.

Seurantajärjestelmä on hyvä lisä tekoälyjärjestelmää - tai oikeastaan mitä tahansa järjestelmää kehitettäessä. Yksinkertaisen seurantajärjestelmän tekeminen on helppoa ja nopeaa, ja siitä saatavat hyödyt ovat merkittäviä. Seurantatyökalut auttavat ongelmien löytämisessä niin kehitettävässä tekoälyssä kuin sen alla toimivassa pelimoottorissakin.

Opinnäytetyössä ei otettu huomioon käytännössä ollenkaan tekoälyn suorituskykyä. Kehitetyn tekoälyn avulla voitaisiin kyllä tehdä tekoälyt useille eri hahmoille, mutta kaikilla hahmoilla olisi omat kappaleensa käytöspuusta. Jatkotutkimusta voisi tehdä siitä, kuinka tekoäly saadaan tehoiltaan rajoitetulla alustalla, kuten mobiililaitteella, toimimaan mahdollisimman monilla hahmoilla. Toisaalta voitaisiin ylipäätään tutkia, kuinka opinnäytetyön kaltainen käytöspuu toimisi ryhmäkäyttäytymistä osaavan tekoälyn pohjana.

LÄHTEET

Kirjallisuus

Baillie-de Byl, P. 2004. Programming Believable Characters for Computer Games. Hingham, Massachusetts: Charles River Media

Berger, L. 2002. Scripting: Overview and Code Generation. Teoksessa Rabin, S. (toim.) AI Game Programming Wisdom. Hingham, Massachusetts: Charles River Media

Champandard, A. 2008. Getting Started with Decision Making and Control Systems. Teoksessa Rabin, S. (toim.) AI Game Programming Wisdom 4. Boston: Course Technology

Kirby, N. 2010. Introduction to Game AI. Boston: Course Technology

Lidén, L. 2004. Artificial Stupidity: The Art of Intentional Mistakes. Teoksessa Rabin, S. (toim.) AI Game Programming Wisdom 2. Hingham, Massachusetts: Charles River Media

Pfeifer, B. 2008. Creating Designer Tunable AI. Teoksessa Rabin, S. (toim.) AI Game Programming Wisdom 4. Boston: Course Technology

Schwab, B. 2004. Ai Game Engine Programming. Boston: Charles River Media.

Tozour, P. 2002 a. The Evolution of Game AI. Teoksessa Rabin, S. (toim.) AI Game Programming Wisdom. Hingham, Massachusetts: Charles River Media

Tozour, P. 2002 b. Building an AI Diagnostic Toolset. Teoksessa Rabin, S. (toim.) AI Game Programming Wisdom. Hingham, Massachusetts: Charles River Media

Verkkolähteet

Champandard, A. 2007 a. On Finite State Machines and Reusability. http://aigamedev.com/open/article/fsm-reusable/ (luettu 8.8.2013)

Champandard, A. 2007 b. The Gist of Hierarchical FSM. http://aigamedev.com/open/article/hfsm-gist/ (luettu 8.8.2013)

Champandard, A. 2007 c. Understanding Behavior Trees. http://aigamedev.com/open/article/bt-overview/ (luettu 8.8.2013)

Hecker, C. 2009. My Liner Notes for Spore/Spore Behavior Tree Docs. http://chrishecker.com/My_liner_notes_for_spore/Spore_Behavior_Tree_Docs (luettu 16.8.2013)

Houghton, D. 2012. Why Street Fighter is still the most important fighting game series around. http://www.gamesradar.com/why-street-fighter-is-still-the-most-important-fighting-game-series-around/ (luettu 31.7.2013)

Isla, D. 2005. CDC 2005 Proceeding: Handling Complexity in the Halo 2 AI. http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php?page=1 (luettu 16.8.2013)

Kasavin, G. 2004. Halo 2 Review. http://www.gamespot.com/halo-2/reviews/halo-2-review-6112628/ (luettu 16.8.2013)

Knafla, B. 2011. Introduction to Behavior Trees. http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/ (luettu 8.8.2013)

Ocampo, J. 2008. Spore: Galactic Edition Review. http://www.ign.com/articles/2008/09/08/spore-galactic-edition-review?page=1 (luettu 23.8.2013)

Tuttle, M. 2011. Finite State Machines for Game Developers. http://matttuttle.com/2011/02/finite-state-machines-for-game-developers/ (luettu 28.10.2013